Artificially Intelligent Foraging

Submitted by Daniel Chalk, to the University of Exeter as a thesis for the degree of Doctor of Philosophy in Biosciences, December 2009.

I certify that all material in this thesis which is not my own work has been identified and that no material has previously been submitted and approved for the award of a degree by this or any other University.

(signature) .....

Abstract

Bumble bees (*bombus spp.*) are significant pollinators of many plants, and are particularly attracted to mass-flowering crops such as Oilseed Rape (*Brassica Napus*), which they cross-pollinate. *B. napus* is both wind and insect-pollinated, and whilst it has been found that wind is its most significant pollen vector, the influence of bumble bee pollination could be non-trivial when bee densities are large. Therefore, the assessment of pollinator-mediated cross-pollination events could be important when considering containment strategies of genetically modified (GM) crops, such as GM varieties of *B. napus*, but requires a landscape-scale understanding of pollinator movements, which is currently unknown for bumble bees.

I developed an *in silico* model, entitled HARVEST, which simulates the foraging and consequential inter-patch movements of bumble bees. The model is based on principles from Reinforcement Learning and Individual Based Modelling, and uses a Linear Operator Learning Rule to guide agent learning. The model incoproates one or more agents, or bees, that learn by 'trial-and-error', with a gradual preference shown for patch choice actions that provide increased rewards.

To validate the model, I verified its ability to replicate certain iconic patterns of bee-mediated gene flow, and assessed its accuracy in predicting the flower visits and inter-patch movement frequencies of real bees in a small-scale system. The model successfully replicated the iconic patterns, but failed to accurately predict outputs from the real system. It did, however, qualitatively replicate the high levels of inter-patch traffic found in the real small-scale system, and its quantitative discrepancies could likely be explained by inaccurate parameterisations. I also found that HARVEST bees are extremely efficient foragers, which agrees with evidence of powerful learning capabilities and risk-aversion in real bumble bees.

When applying the model to the landscape-scale, HARVEST predicts that overall levels of bee-mediated gene flow are extremely low. Nonetheless, I identified an effective containment strategy in which a 'shield' comprised of sacrificed crops is placed between GM and conventional crop populations. This strategy could be useful for scenarios in which the tolerance for GM seed set is exceptionally low.

Acknowledgements

I wish to thank both of my supervisors, Dr. James Cresswell and Prof. Richard Everson, for their unwavering support and guidance throughout the project, both on a professional and a personal level. I thank the BBSRC (Biotechnology and Biological Sciences Research Council) for funding this project. I am grateful for the support of the university's Biosciences Glasshouse Technician, James Chidlow, for helping to set up the empirical studies I conducted, and to all of the students who assisted with the empirical observations (Abul Al-Azad, Timothy Paulden, Barry Evans, Rebecca Watson, David Hannaford and Ossama Elmenyawy). I would also like to acknowledge my late parents, who offered me an extraordinary amount of love and support when I was growing up, inspiring me to strive to succeed throughout my life; I dedicate this thesis in memory of them both.

Table of Contents

Abstract 2

Acknowledgements 3

Table of Contents 4

List of Figures 9

List of Tables 13

List of Equations 14

List of Parameters 15

Definitions 16

Abbreviations 17

Chapter One : Bumble Bees – Landscape-Scale Foragers and Pollen Vectors 18

1.1 Foraging 18

1.1.1 Foraging behaviour 18

1.1.2 The influence of foraging behaviours in ecological systems 20

1.2 The study of animal behaviour 21

1.2.1 Studying animal behaviour empirically 21

1.2.2 Studying animal behaviour theoretically 22

1.3 A contemporary application : GM containment and bee-mediated cross-pollination in crops 25

1.3.1 Bees, foraging and polli nation of plants 25

1.3.2 GM crops – cross-pollination and containment 27

1.3.3 How to determine the threat to GM containment posed by bee-mediated cross-pollination? 28

1.3.4 Predicting pollinator mediated gene flow theoretically : the E-Psi-b model 30

1.4 The assumption of optimality 32

1.4.1 An economics-based analysis of bumble bee movements 32

1.4.2 Optimal Foraging Theory 33

1.4.3 Possible criticisms of the assumption of optimality 34

1.4.4 The foraging ‘problem’ for bumble bees 37

1.5 Why we cannot use existing approaches 37

1.5.1 Why we cannot use the ‘Ideal Free Distribution’ theory 37

1.5.2 Why we cannot use the Marginal Value Theorem 38

1.5.3 Why we cannot use the Threshold Departure Rule (and other ‘rules of thumb’) 39

1.5.4 Why we cannot use Job Search theory 40

1.5.5 Why we cannot use HOOFS 41

1.6 Learning 42
1.6.1 Towards a learning-based approach 42
1.6.2 Learning and memory in bees 43
1.7 Existing learning approaches 44
1.7.1 Why we cannot use an existing learning-based IBM approach 44
1.7.2 Why we cannot use the Bayesian approach 45
1.7.3 A promising solution - Reinforcement Learning 46
1.8 Summary 48
Chapter Two : HARVEST – An AI Foraging Model 50
2.1 Introduction 50
2.2 Economically motivated foraging among patches 51
2.3 Reinforcement Learning 51
2.4 Formulating the first principles of bumble bee behaviour 52
2.5 HARVEST 54
2.5.1 Description of the model 54
2.5.2 Measuring foraging performance 63
2.5.3 Calculating bee-mediated gene flow 64
2.6 Validation of performance 65
2.7 A contemporaneous application of the RL approach by others 68
2.8 Summary 69
Chapter Three : Paper – A reinforcement learning solution to economically-motivated patch choice : application to landscape-scale bumble bee foraging 71
Chapter Four : Recreation of Iconic Patterns of Bee-Mediated Gene Flow by HARVEST 99
4.1 Introduction 99
4.2 Investigation (i):Patch scale optimisation of 'Memory Parameter', β 100
4.2.1 Introduction to Investigation (i) 100
4.2.2 Methods 101
4.2.3 Results 102
4.2.4 Discussion 103
4.3 Investigation (ii): Landscape-scale optimisation of the 'Memory Parameter', β 105
4.3.1 Methods 105
4.3.2 Results 107
4.3.3 Discussion 107
4.4 Investigation (iii): Iconic Pattern 1 : Increasing inter-patch distance decreases gene flow levels 108
4.4.1 Introduction to Investigation (iii) 108

4.4.2 Methods 111
4.4.2.1 Patch Classifications 111
4.4.2.2 The 'Spatially-Explicit' Landscape 111
4.4.2.3 Solving the E-Psi-b model 112
4.4.2.4 General Parameterisations (Consistent for all Experiments) 112
4.4.2.5 Experiment IP1.1 115
4.4.2.6 Experiment IP1.2 116
4.4.2.7 Experiment IP1.3 116
4.4.3 Results 118
4.4.3.1 Experiment IP1.1 118
4.4.3.2 Experiment IP1.2 120
4.4.3.3 Experiment IP1.3 122
4.5 Investigation (iv): Iconic Pattern 2 : Increasing sink patch size decreases gene flow levels 123
4.5.1 Introduction to Investigation (iv) 123
4.5.2 Methods 125
4.5.2.1 Experiment IP2.1 127
4.5.2.2 Experiment IP2.2 128
4.5.2.3 Experiment IP2.3 129
4.5.3 Results 130
4.5.3.1 Experiment IP2.1 130
4.5.3.2 Experiment IP2.2 131
4.5.3.3 Experiment IP2.3 132
4.6 Discussion of Iconic Pattern Replication 133
4.7 Summary 136
Chapter Five : An Empirical Test of HARVEST 137
5.1 Introduction 137
5.2 Methods 138
5.2.1 The foraging landscape 138
5.2.2 Data capture 138
5.2.3 Quantifications of key parameters 141
5.2.4 Nectar production 142
5.2.5 Bee foraging rate 142
5.2.6 Parameterization of the model 143
5.2.7 Implementation of HARVEST 146
5.3 Results 147
5.3.1 Empirical study results 147

5.3.2 Model results 151
5.4 Sensitivity analysis of results to variation in the number of simulated bees 154
5.4.1 Sensitivity analysis of results to variation in bee abundance 154
5.4.2 Sensitivity analysis of results to variation in nectar Replenishment rate 155
5.5 Discussion 157
5.5.1 Insights into bee foraging behaviour 157
5.5.2 Prediction of observed foraging movements 158
5.5.2.1 Patch-to-patch movements, overall traffic (SWIPT) and traplining 158
5.5.2.2 Residence and arrival rates 159
5.5.3 Traplining 159
5.6 HARVEST vs observations – overall comments 160
Chapter Six : Landscape-Scale Bee-Mediated Gene Flow and Containment 162
6.1 Introduction 162
6.2 Parameterisation of the model for the landscape-scale 163
6.3 Investigation (i) : Landscape-scale behavioural exploration 166
6.3.1 Methods 166
6.3.1.1 Experiment LS1.1 166
6.3.1.2 Experiment LS1.2 166
6.3.2 Results 167
6.3.2.1 Experiment LS1.1 167
6.3.2.2 Experiment LS1.2 168
6.3.3 Discussion 169
6.3.3.1 Finding (a) : why does SWIPT increase with decreasing variability in Q? 169
6.3.3.2 Finding (b) : why does SWIPT increase with increasing L? 170
6.3.3.3 Consequences for gene flow 171
6.4 Investigation (ii) : The Sacrificial Shield as a strategy for gene containment 172
6.4.1 Introduction to Investigation (ii) 172
6.4.2 Methods 173
6.4.2.1 General Methods 173
6.4.2.2 Experiment LS2.1 173
6.4.2.3 Experiment LS2.2 174
6.4.2.4 Experiment LS2.3 175

6.4.2.5 Experiment LS2.4 176
6.4.3 Results 177
6.4.3.1 Experiment LS2.1 177
6.4.3.2 Experiment LS2.2 178
6.4.3.3 Experiment LS2.3 179
6.4.3.4 Experiment LS2.4 180
6.4.4 Discussion 180
6.5 Summary 183
Chapter Seven : Implications and Further Work 185
7.1 Introduction 185
7.2 Implications 185
7.2.1 Gene flow levels in landscapes 185
7.2.2 Containment strategies 186
7.2.3 Iconic patterns 187
7.2.4 Efficient foraging despite incomplete information 188
7.2.5 Foraging behaviour at the small scale 188
7.2.6 Critical evaluation of the model's predictions 189
7.3 Suggestions for further Work 191
7.3.1 Incorporation of systematic foraging and traplining 191
7.3.2 Incorporation of continuous nectar distributions 193
7.3.3 Specifying spatially-explicit flowers within patches 194
7.4 General Summary 195
Appendix A : Transition Matrix Extraction – Worked Example 196
Appendix B : Redundancy of the Non-Greedy Exploration Parameter 199
Appendix C : HARVEST Simulation Screenshots 202
Appendix D : HARVEST Code Listing 208
Bibliography 285

List of Figures

Figure	Description	Page
1.1	Key components of the E-Psi-b model.	31
2.1	Flow Chart showing the foraging process of a bee within the HARVEST model.	61
2.2	Percentage of a bee's encounters with full flowers in a patch with varying patch qualities compared to the expected percentage of full flower encounters after 1000 foraging bouts of foraging activity.	66
2.3	System-wide inter-patch traffic (SWIPT) with variable C and L.	68
C3 (figure 1)	The cumulative number of unique fields visited by a bee (y-axis) vs. the number of time steps elapsed (x-axis).	93
C3 (figure 2)	Mean number of field-to-field transitions per bee per foraging bout (y-axis) vs. variation in quality among fields, G*, (x-axis) for various sizes of landscape.	94
C3 (figure 3)	Mean number of field-to-field transitions per bee per bout (y-axis) vs. landscape richness (x-axis).	95
C3 (figure 4)	Contour plot showing the impact on foraging performance levels of resource variability, G*, (expressed as the range between the highest and lowest quality fields) and bee capacity (or landscape richness).	96
C3 (figure 5)	(a) Mean proportion of flowers (or seed set) of a gene flow-sink field completely fertilized by incoming pollen (y-axis) vs. variation in quality among fields, G* (x-axis). (b) Mean probability of arriving in the sink field from a source field, E (y-axis) vs. variation in quality among fields, G* (x-axis). (c) Mean number of flowers pollinated by bees during a Visit to the sink field, b, (y-axis) vs. variation in quality among fields, G* (x-axis).	97
C3 (figure 6)	Landscape-scale mean foraging performance per bee per foraging bout (E(x)) per trial with varying β and varying landscape reward variability.	98
4.1	Mean foraging performance per bee per foraging bout per trial in small-scale landscapes, with varying β, varying landscape size (number of patches) and varying economic richness.	102
4.2	Mean inter-patch traffic per bee per foraging bout per trial in a landscape with six patches, varying β and varying economic richness (1 : 1 rich : poor replenishment frequency ratio, up to 4 : 1).	103
4.3	Landscape-scale mean foraging performance per bee per foraging bout (P(x)) per trial with varying β and varying landscape reward variability.	107
4.4	Examples of the iconic gene-flow distance relationship.	109

4.5	Summary of landscape configurations used for investigation (iii).	115
4.6	Experimental configuration and results for experiment IP1.1 testing for iconic pattern 1.	118
4.7	Experimental configuration and results for experiment IP1.2 testing for iconic pattern 1.	120
4.8	Experimental configuration and results for experiment IP1.3 testing for iconic pattern 1.	122
4.9	An example of the typical pattern of the patch-size gene-flow relationship.	123
4.10	Summary of landscape configurations used for investigation (iv).	127
4.11	Experimental configuration and results for experiment IP2.1 testing for iconic pattern 2.	130
4.12	Experimental configuration and results for experiment IP2.2 testing for iconic pattern 2.	131
4.13	Gene flow levels from source to sink with increasing size of sink and varying rate of replenishment.	133
5.1	Mean nectar production rates (microlitres h-1) in flowers of droughted (‘D’) and watered (‘W’) individuals of B. napus under glasshouse conditions.	142
5.2	The frequency of total successive patch visits for during a single foraging bout on the array in the sample observation session.	147
5.3	Mean patch residence (in seconds), arrival rates, and overall bee effort, comparing homogenous and variable landscapes, and W and D patches.	148
5.4	Frequency of specific inter-patch movements made by real bees in the final observation session of the empirical study.	150
5.5	Mean patch residence (in flower visits), arrival rates, and overall bee effort, comparing empirical data and model predictions.	151
5.6	Mean number of consecutive patches visited by bees upon visiting the array.	152
5.7	Frequency of specific inter-patch movements made by HARVEST bees in a sample homogenous trial.	153
5.8	Frequency of specific inter-patch movements made by HARVEST bees in a sample variable trial.	154
5.9	Residence and arrival results for the empirical comparison that include HARVEST’s simulation of just a single bee foraging in the grid.	155
5.10	Mean patch residence per bee with varying replenishment intervals tested in a homogenous landscape system in the model.	156

5.11	Mean total patch visits per bee (arrival data) with varying replenishment intervals tested in a homogenous landscape system in the model.	157
6.1	Illustration of how the landscape configuration is altered with each incremental increase of landscape size tested.	166
6.2	a) The mean number of flowers visited per bee per field, with varying landscape size L. b) The mean number of inter-patch movements per bee per 8 hour foraging trial, with varying landscape size L. c) Mean foraging efficiency per bee per foraging bout, with varying landscape size L.	167
6.3	a) The mean number of flowers visited per bee per field, with varying range in field qualities. b) The mean number of inter-patch movements per bee per 8 hour foraging trial, with varying range in field qualities. c) Mean foraging efficiency per bee per foraging bout, with varying range in field qualities.	168
6.4	Configuration of the landscape and the shield in Sacrificial Shield Experiment LS2.1.	173
6.5	Configuration of the landscape and the shield in Sacrificial Shield Experiment LS2.2.	174
6.6	Configuration of the landscape and the shield in Sacrificial Shield Experiment LS2.3.	175
6.7	Configuration of the landscape and the shield in Sacrificial Shield Experiment LS2.4.	176
6.8	Gene flow, expressed as the percentage of GM pollen deposited by bees in the Conventional county, with varying grades of prominence of the shield.	177
6.9	Gene flow, expressed as the percentage of GM pollen deposited by bees in the Conventional county, with varying distance of the shield from the GM county.	178
6.10	Gene flow, expressed as the percentage of GM pollen deposited by bees in the Conventional county, with varying distance between the GM and Conventional counties.	179
6.11	Gene flow, expressed as the percentage of GM pollen deposited by bees in the Conventional county, with varying quality of the fields in the shield.	180
B1	Mean foraging performance per bee with varying values of ε and β.	200
C1	Main configuration window of HARVEST.	202
C2	Window that provides ability to specify precise patch sizes for each	203

patch in the landscape.
C3 Window that provides ability to specify exact coordinates for patches within the landscape, rather than using the default matrix-style configuration. 203
C4 Screen capture of HARVEST when the simulation is running. 204
C5 Main results window. 204
C6 Window allowing read-only access to the transition matrix file. 205
C7 Window showing breakdown of residence and visit numbers to each individual patch in the landscape. 205
C8 Window showing report of how often bees that returned to the nest became ‘naive’ via the Ω parameter. 206
C9 Window that allows the user to watch the movements that bees made within the landscape in the simulation. 207

List of Tables

Table	Description	Page
2.1	Summary of parameters used by the model	62
4.1	Summary of parameterisation for investigation (iii)	114
4.2	Summary of parameterisation for investigation (iv)	126
5.1	Average number of open flowers per patch on each day of the Empirical study, along with the type of patch (W or D) to which the flower count relates	141
5.2	Summary of parameterisation for the model's replication of the empirical experiment	146
6.1	Summary of general parameterisation for landscape-scale experiments	165

List of Equations

Equation	Description	Page
1.1	E-Psi-b model of pollinator-mediated gene flow	31
2.1	HARVEST Linear Operator Learning Rule	57
2.2	HARVEST action-value calculation	59
2.3	HARVEST foraging efficiency calculation	63
2.4	E-Psi-b model, re-written in terms of HARVEST parameters	64
C3 (eq 1)	Rate of gain equation	77
C3 (eq 2)	HARVEST action-value calculation	78
C3 (eq 3)	HARVEST Linear Operator Learning Rule	78
C3 (eq 4)	E-Psi-b model	81
C3 (eq 5)	Approximate highest rate of gene flow calculation	81
5.1	Replenishment interval estimation assuming systematic foraging	144
5.2	Estimation of simultaneous bee visitors in empirical study	145

List of Parameters

Parameter	Identifier
ξ	Proportion of sink patch's seed with source patch paternity mediated by cross-pollination
E	Fraction of pollinators arriving at sink from source
Ψ	Number of fruits fully fertilised in sink with source pollen by each pollinator in a sink visit
b	Total sink fruits fertilised by each pollinator in a sink visit
B	Number of bees in the grid
B_E	Estimated number of bees in the grid
L	Number of patches in landscape
Q	Initial quality of patches
Q(i)	Actual quality of patch i
q(i)	Estimated quality of patch i
β	Sensitivity to individual flower sample
s(i)	Value of the reward from latest sample in patch i
v(i, j)	Action-value for visiting patch j when currently in patch i
C	Bee's nectar carrying capacity
c	Bee's remaining nectar carrying capacity
h	Flower handling time
t(i, j)	Flight time between patches i and j
F_i	Number of flowers in patch i
I_i	Replenishment interval for patch i
P(x)	Bee's foraging efficiency in foraging bout x
G(x)	Elapsed time on the global clock since foraging bout x began
R_{full}	Reward (in nectar units) offered by a full flower
R_{empty}	Reward (in nectar units) offered by an empty flower
T	Total time spent observing in empirical study
T_B	Total time (in seconds) spent actively observing bee activity in empirical study
a	Proportion of array observed during an empirical study session

Definitions

Dry (D) Patch	A patch of flowers that is droughted
Floral Catchment	The set of patches representing the surrounding landscape external to the sink and source populations.
HARVEST	**H**arvesting **A**nimal **R**einforced **V**alues and **Est**imates. A Reinforcement Learning-based foraging model.
Omniscient Bee	A HARVEST bee that is always aware of the true quality of patches in the landscape
Performance	The quantified foraging efficiency of a HARVEST bee, expressed in terms of the amount of resource gathered divided by the time taken to do so
Sacrificial Shield	An array of resource sites whose genetic purity is sacrificed to preserve the genetic purity of another population.
SWIPT	System-wide inter-patch traffic. A measure of the level of movement by bees on average within the foraging system.
Wet (W) Patch	A patch of flowers that is kept healthy by being regularly watered

Abbreviations

%	Percentage
µ	Micro
AI	Artificial Intelligence
ANN	Artificial Neural Network
B. napus	*Brassica napus*
BL	*Bombus lapidaries*
BP	*Bombus pratorum*
BT	*Bombus terrestris*
D	Dry
DEFRA	Department for Environment, Food and Rural Affairs
EU	European Union
GM	Genetically Modified
h	Hours
HARVEST	Harvesting Animal Reinforced Values and Estimates
HOOFS	Hierarchical Object Oriented Foraging Simulator
IBM	Individual Based Model
IFD	Ideal Free Distribution
IP	Iconic Pattern
JST	Job Search Theory
km	Kilometre
l	Litre
LS	Landscape-Scale
m	Metre
MVT	Marginal Value Theorem
OFT	Optimal Foraging Theory
PMF	Plant Molecular Pharming
RL	Reinforcement Learning
RNEI	Rate of Net Energetic Intake
s	Seconds
S.E.	Standard Error
SS	Sacrificial Shield
t.u.	Time Unit
TDR	Threshold Departure Rule
W	Wet

Chapter One : Bumble Bees - Landscape-Scale Foragers and Pollen Vectors

1.1 Foraging

1.1.1 Foraging behaviour

The behavioural traits of animals can vary enormously, and the study of animal behaviour is a significant area of classical and contemporary science (e.g. Skinner, 1936; Lorenz, 1966; Frisch, 1950; Tinbergen, Impekoven & Franck, 1967). Despite a rich diversity in behaviour among species, there are notable threads of commonality that link many animals, such as the need for all animals to obtain resources for growth and reproduction. Resources are typically essential to the animal seeking them, and include the food required for an animal to survive and grow. They may be derived from plants or other animals. For social animals, resources may also be necessary for the survival and development of the group with which the individual animal lives.

Often animals must search for their food and this process is termed *foraging*. The problem of where to forage for food is non-trivial, and here I consider five key influences on foraging behaviour. First, the food within an animal's foraging environment is likely to be patchily distributed (Ricketts, 2001; Westphal *et al*, 2006; Kohlmann and Risenhoover, 1998; Klaassen *et al*, 2006; Iwasa *et al*, 1981) with some areas offering more food than others. Animals foraging for floral rewards such as nectar and pollen may encounter spatial variability in the amounts of resources in flowers, or 'standing crop levels' (Westphal *et al*, 2006), and like any predators hunting for animal prey, they will need to focus their efforts where their 'prey items' are more likely to be most abundant and rewarding. Second, as well as spatial variation, floral foragers often encounter temporal variation as the levels of food availability change over time. Variation may be the result of food depletion by competing foragers (Inouye, 1978) or environmental factors such as the transience of peak bloom in plants (Heinrich, 1976b), as well as replenishment mechanisms that cause increases to food levels, such as the birth of new prey animals or plants regenerating floral rewards to attract pollinator activity (Klinkhamer and de Jong, 1990). In the case of animal prey, dispersal and migratory events and the avoidance of predators will cause the location of the predator's food to move over time, but this is not relevant for floral foragers, whose 'prey' is sessile.

Third, energetic costs and benefits of alternative foraging activities must be assessed by the animal in order that is able to forage efficiently. Animals may be under selection to maximise returns on foraging efforts, and so the decision of where to forage is therefore influenced by the inherent compromises between the time and effort required to reach and forage at a source of food, and the quantity and quality of the food that can be obtained therein (Stephens and Krebs, 1986). In addition, foraging decisions may depend on the animal's internal state (Mangel and Clark 1986). For example, an animal close to starvation is likely to choose a closer foraging site rather than making a longer journey which it may not survive. Fourth, predators may themselves be the prey of other predators. Whilst the acquisition of food is likely a dominant motivator of many animals, they must also consider the risks of being attacked by predators, lest the seeking of food for survival be in vain. In situations where there is a predation threat, the movement dynamics of foraging animals are likely to be influenced by predator avoidance mechanisms (Werner *et* al, 1983; Gilliam and Fraser, 1987).

Fifth, the behaviour of a forager may be influenced by its need to have a fixed shelter or home. In this case the animal is termed a central place forager. A central place forager returns to a central place – typically its home – when it has collected sufficient resources (Pyke, 1984). This may be because it needs to supply the resources to others within a social group or because it needs to store the resources safely for future consumption (Rubetra, 1981). Central place foraging will affect foraging behaviour because the animal must take into account the time and energy required to make trips to and from its central place (Andersson, 1978). This influence may not be limited to generating bias towards sites that are closer to the central place, and for social animals may even result in the inverse effect, as the area surrounding a central place may be particularly depleted due to the density of foragers in that area that share the animal's home, or 'central place' (Dramstad, 1996).

In sum, it seems that foraging animals typically face a surprisingly complex problem when making decisions about where and when to forage. Despite these complexities, however, animals have time and again shown themselves to be very efficient foragers (Stephens and Krebs, 1986). This makes the study of foraging behaviour an inherently interesting area of research, as attempts are made to explore how animals implement decision-making methods to solve their foraging problems, and how their decisions influence their movement behaviours. As I show within this chapter, however, the relevance of studying foraging behaviour extends beyond simple curiosity; the movement dynamics of foragers can have direct impacts upon other ecological systems within and external to their foraging environment. Therefore, foraging

theory is a vital area of study with which we may obtain a solid grasp of the impact of foraging dynamics within various problem domains.

1.1.2 The influence of foraging behaviours in ecological systems

Foraging processes influence movement dynamics in animals and consequently they can have an impact upon a range of other animal behaviours. For example, dispersal mechanisms describe the way in which subsets of animal populations break away and move into new areas of the environment. Dispersal processes typically occur as the result of some form of change in the animal's environment. Areas may become depleted of food supplies, leading to animal groups moving to find locations within the foraging environment where they are able to maintain acceptable rates of food intake (Waser, 1985). Alternatively there may be changes in food patterns across the landscape, such as the dispersal or migration of animal prey, necessitating moves to follow patterns of food availability (Holyoak and Lawler, 1996).

A wide variety of bird, insect and fish species exhibit migratory behaviours (Dingle and Drake, 2007), which can emerge in response to changes in food availability as animals move to new foraging environments that have increased levels of food availability or preferred sources of food. We can therefore see that pivotal influences on foraging behaviour such as food availability, hunger levels and diet optimization considerations, can be driving forces behind migration patterns and, ultimately, on patterns of dispersion and distribution (Tyler and Hargrove, 1997).

The study of animal behaviour also finds important application in the assessment of measures to conserve animal species and retain biodiversity. Significant losses to biodiversity are very real threats, and the impact of the loss of one species can cause consequences for other species and ecosystems (Conner *et al*, 2003). Measures to conserve species are desirable, though the solutions may not always be obvious. For example, wild bee species worldwide are under threat (Williams and Osborne, 2009). Colony Collapse Disorder (Williams, 2008) is emerging as a significant concern for the preservation of honey bee colonies and notable declines in bumble bee numbers have been observed (Goulson *et al*, 2008; Goulson *et al*, 2005). Such declines may be caused by changes to agricultural practices, the removal of many suitable wild areas for their nests to be built and declines in feeding sites of wild flowers containing much needed nectar and pollen reserves (Goulson *et al*, 2008). One approach to mitigating such worrying

declines is to identify foraging landscapes that promote sustainable populations of bees and to deploy such configurations within the framework of the modern agricultural landscape.

1.2 The study of animal behaviour

1.2.1 Studying animal behaviour empirically

By far the most widely adopted approach to the study of animal behaviour is the empirical approach. Here, animals are studied either within their natural environments or within laboratory environments that attempt to recreate aspects of the real-world, but either way the target organism is studied directly. The results obtained are direct observations of the behaviour of the animals themselves. Since the ultimate aim of any animal behaviour study is to learn more of the behaviour of a given animal, then direct observation should, in principle, yield the most accurate results. Traditionally, the empirical study of animal behaviour has come from two distinct areas of science – *Behavioural Ecology* and *Experimental Psychology* (Houston, 2009). Behavioural ecology approaches are typically characterised by experiments that take place in the natural habitats of the animals being studied (e.g. Lilliendahl, 1998; van der Veen, 1999; Goss-Custard *et al*, 1995). This methodology has the obvious advantage of yielding directly relevant results as the behaviours observed are contextualised by the natural environment of the animals, lending immediate credence to the behaviours being observed. That is not to say that such results can always be considered 'pure' and free of bias however; the mechanisms employed to capture behavioural data could arguably affect the very behaviour being studied, adding a sense of doubt to conclusions. By studying in the wild, the experimenter may also lack the ability to sufficiently control conditions to test for specific behaviours or hypotheses. Experimental psychology approaches, being traditionally laboratory-based (e.g. Krebs *et al*, 1977; Neuman *et al*, 1997; Cowie, 1977), do not tend to suffer from this problem, but by their nature run the risk of inaccurately recreating real world conditions and phenomena, thereby eliciting artificial behaviours.

Thus, whilst the empirical approach represents a clear 'ideal', a number of practical factors mean that empirical testing is not always appropriate or even feasible. In order to be able to study animals empirically, there must be at least a means of observing the animal and a means of measuring its behaviour. For slower animals that are easily spotted and that have a limited movement range, observation practices may be very simple. However, when studying animals that are faster, more difficult to spot (perhaps because of size or other physical traits) and that move over large distances, the problem of how to observe them may be problematic. For example, bumble bees (*Bombus spp.*) are typically small and difficult to track, because they can

move at a speed of approximately 7 ms^{-1} and can travel over kilometre distances from their nests (Osborne *et al*, 1999; Westphal *et al*, 2006). In such cases, heavier reliance must be placed on technology to track and capture data from animals, but such technologies can come with limitations that lead to decreased accuracy and vexing gaps in the acquired data set, as I show later.

There are additional obstacles in the empiricist's programme. For example, empirical experiments necessarily incur costs in both money and time. The cost of acquiring and developing technologies required for the experiment, costs of analysing data, remuneration of participants and other running costs must all be considered when planning an empirically-based investigation within the typical constraints of limited resources. Furthermore, for many experiments, it is possible that interference is caused to the studied animal simply by conducting the experiment and, therefore, the observed behaviours may not be truly representative. Additionally, in some ecological scenarios the window of opportunity for observations may be relatively narrow – observations may need to coincide with the peak bloom of a plant for example. Capturing a meaningful dataset can be an impractical target when studying certain animals empirically. Whilst I am by no means suggesting that empirical methods lack credibility, it is worth highlighting that in some instances, empirical investigation is not without severe limitations. In such cases, a theoretical approach may be a pragmatic proxy for the empirical programme, and I will propose that the landscape-scale study of bumble bees is a case in point.

1.2.2 Studying animal behaviour theoretically

Theory-based approaches are a widely-used alternative to empirical study. Unlike the empirical approach, theoretical studies do not involve directly observing the behaviours of target organisms. Instead, the animal behaviour is represented using an abstraction, typically a mathematically formal analogy, and is thereby explored as a model either in mathematical algebra (Emlen, 1966; Charnov, 1976; Schoener, 1971), or as an *in silico* simulation (Beecham and Farnsworth, 1998; Bernstein, Kacelnik & Krebs; 1988), which in its simplest form will track the performance of a 'rule of thumb' (Hodges, 1985). Typically, models do not try to capture all of the potential behavioural aspects of a particular animal (which would be impractically too large in scope), but instead focus on only a subset of the animal's properties that capture the essence of the behaviour(s) being investigated (Stephens & Krebs, 1986).

Mathematical models use equations and functions to generate outputs that represent predictions of animal behaviour, based upon inputs that characterise the parameters of the scenario. For example, a mathematical model that predicts the food type(s) upon which an animal should forage - commonly known as the 'diet choice problem' (Stephens and Krebs, 1986) - may take amongst its inputs the quantified availability of each food type, along with the hunger level of the animal. The output may represent a threshold level of 'reward quality', with food items above the threshold predicted to be accepted by the animal and all others to be rejected. Typically, the model is tuned to optimize the value of the output variable(s), which mimics the effect of natural selection on a fitness-related beviour (Stephens & Krebs, 1986). In general terms, the mathematical approach outputs a numerical solution which can be interpreted to determine when certain behaviours will arise.

Models are not created in conceptual isolation, however, and typically draw on observations and results from the empirical research tradition. For example, whilst both approaches have their strengths and weaknesses, the inherent commonality in their objectives would suggest the strongest studies should consider elements from both approaches (Houston, 2009). Principles of reinforcement are a recurring theme within the psychological literature (Skinner, 1938; Lea, 1979; Mazur, 1984; Kacelnik and Bateson, 1996; Neuman *et al*, 1997; Keasar *et al*, 2002). The idea is that animals adjust their behaviour according to how they are rewarded when they choose certain actions. Rewards can vary both in terms of their magnitude and their frequency (or likelihood). Time and again, animals have been shown to be extremely responsive to reinforcement schedules, suggesting that the proposition that 'actions are reinforced by feedback' is an expression of a fundamental (and realistic) cognitive element, at least in relation to the decision-making mechanisms that underpin foraging.

Indeed, we can usefully consider behavioural dynamics as being emergent results of decision-making processes. An animal has a set of decisions to make, and a set of ways in which it can make those decisions. Only a subset of the possible approaches to modelling decision-making will be suited for any given decision. We can perhaps hope that evolutionary processes have likely favoured organisms that employed decision making processes that were simpler, not only because of the limited cognitive capacity of many animals, but also because simpler rules tend to be more efficient. It has been shown that the employment of simple heuristics (or 'rules of thumb') can yield surprisingly effective results that often compete with (or even outperform) far more complex decision making rules (Todd, 2000). Otherwise, in constructing a model, we must necessarily invoke the principle of parsimony. In any event, the principles of simplicity

and parsimony must be an important guiding consideration when studying animal behaviour in a theoretical capacity.

Rules of thumb are simple protocols that could be followed by an animal to determine how decisions should be made in given situations without recourse to advanced mathematical computation. To use the diet choice example, a rule of thumb might simply state 'select an item of the prey type that you have consumed before over any prey type you haven't' – an example from the class known as Ignorance-Based Decision Heuristics (Todd, 2000). The idea behind a rule of thumb is to provide a simple and unambiguous output based on a simple algorithm that draws on sensory inputs.

In-silico simulation is a relatively young sub-domain of theoretical animal behaviour modelling, that has grown alongside the developments in computational power, and can be extended to problems that have to many constraints to be construed (and solved) in simple algebra. The most obvious advantage of using computer-based simulation is the ability to explore a wide range of scenarios with ease, which potentially enables predictions to be generalised to any time and place that the simulation can be adapted to represent. However, the advent of Individual-Based Modelling (IBM) approaches in particular (Grimm and Railsback, 2005) has shown how the technology can be used as more than an automated calculator for existing approaches. Instead, development focuses on how to harness the advantages of the technology to create entirely new approaches that more closely replicate real-world scenarios (Grimm and Railsback, 2005). I discuss the IBM approach later in the chapter, but essentially it allows for individual animals to be simulated as explicit virtual entities, each with a degree of 'intelligence' in the form of behavioural rules that determine how each individual behaves. This is a potentially profitable approach in the study of animal behaviour.

In general terms, theoretical approaches have certain advantages over their empirical counterparts. Once in place, the model allows for a variety of scenarios to be tested swiftly and with ease. Entire landscape configurations, or even animal species, can be interchanged and new results generated rapidly. New theories can easily be tested without the concerns of reorganising field or laboratory experiments. Other than the cost of any technology required to generate calculations from the model, experimental costs are minimal. Also, by not using human observers or technologies to capture data, the capacity for human error or technical issues is significantly minimised. The emphasis of theoretical modelling is the feasible

implementation and control of experiments, which provides a valuable complement to the empirical programme.

A further advantage of the theoretical approach is its ability to uncover knowledge beyond the capabilities of empirical experiments. For example, it is possible to investigate the behaviour of 'stupid' or 'omniscient' animals, which do not exist in reality. Thus, theoretical models can generate predictions for domains in which an equivalent empirical experiment simply is not possible. This makes modelling a powerful means of exploring new areas and making new discoveries, and can help to 'fill the gaps' in empirical data sets, because extreme scenarios can also be tested that may be rare or non-existent in real world ecology, but may be of interest either in themselves, or in identifying behavioural relationships and trends.

Theoretical modelling is not without its own problems. Theoretical approaches generate predictions of behaviour, but they have only limited validity in themselves. Unlike empirical studies in which a real animal is being observed and as such is intrinsically 'valid', or credible, theoretical predictions must be validated by reference to some 'real world phenomena' to determine how accurate they are. Whilst some validation can be attained by logically testing the elements that comprise the model, and arguing the reasonableness of the assumptions and design decisions made, the most convincing validation requires the use of empirical experiments to corroborate theoretical predictions, with the verisimilitude of the theoretical data used as an indicator of the validity of the model, as I do in this study.

1.3 A contemporary application : GM containment and bee-mediated cross-pollination in crops

1.3.1 Bees, foraging and pollination of plants

Bees in themselves are foragers, but their foraging decisions have implications for the plants whose flowers they visit and thereby pollinate. Foraging among flowers is an activity in which each forager obtains discrete resource units found in packages, or flowers (Biesmeijer and Tóth, 1998). Flowers receive visits from a wide array of animals, e.g. hummingbirds, moths, bats and bees (Proctor, Yeo & Lack, 1996), but here I will be primarily considering social Hymenoptera – specifically bumble bees – which have been widely studied (Proctor *et al.*, 1996; Goulson, 2003). Bees collect both pollen and nectar (Goulson, 1999). Pollen provides protein, which is used by social and solitary bees for provisioning offspring (Goulson, 1999), and nectar provides

carbohydrates, and so is used both as an energy source for adults (Goulson, 1999), and, in social bee colonies, for heating the nest (Heinrich, 1979).

Bees are pollen vectors for a variety of plant species (Proctor *et al*, 1996; Klein *et al*, 2007). A pollen vector enables seed set through *pollination*, which is the transfer of pollen from the anthers of a flower to the stigma of the same flower, or another flower. This process typically results in the *fertilization* of the plant. *Cross-pollination* describes the transfer of pollen from the anther of a flower on one plant to the stigma of a flower on another individual plant. Cross-pollination is therefore a prerequisite for *gene dispersal* (Proctor *et al*, 1996). Typically, a flower requires relatively few visits to satisfy its requirements for seed set (Bell, 1985), whereas many more visits are required for complete dispersal of pollen. For example, in oilseed rape (*Brassica napus* L.), bumble bees typically deposit in the region of 200 to 800 pollen grains on a flower's stigma in a single visit (Cresswell, 1999), whilst the flower needs to receive only c. 100 pollen grains to fertilize all the (c. 30) ovules in the ovary (Cresswell *et al*, 2002).

Spatial patterns of cross-pollination will be shaped by the inter-flower movements of pollinators. Pollinator movements arise from their foraging behaviour. Thus, an understanding of the movement dynamics of pollinators is vital for the preservation of plant species (Ellstrand, 1992). Pollinators cause pollination while deliberately seeking pollen and nectar, but these processes can occur as a side-effect of foraging activities, such as a bee's pollen load brushing off onto recipient flower stigmas as it exploits a flower for nectar (Young and Stanton, 1990; Heinrich, 1976), or pollinators that are seeking alternative feeding sites making movements between plant populations leading to cross-pollination (Heinrich and Raven, 1972). Overall, an understanding of the spatial movements of pollinators is valuable, and as a result they have received significant scientific attention, because gene flow can affect the persistence and evolution of plant species (Slatkin, 1985). Furthermore, with the advent of GM crops, pollinators can introduce foreign genes into a conventional plant population (Ellstrand, 1992, Ellstrand, 2001), which may be either a wild relative or a conventional variety of the same crop. This potential for the spread of GM transgene has become a cause of public concern (Dale, 2005; and see below). For the case of GM varieties of domesticated crop plants, I outline such a scenario later in this chapter, which has been a driving force for my research.

1.3.2 GM crops – cross-pollination and containment

Genetically modified (GM) crops are crops that have been altered by genetic engineering in order to enable the expression of novel genes within the plant (Nap *et al*, 2003). Modifications may also be made to provide or augment herbicide and pesticide-resistant properties in plant species, so as to improve the efficiency of agricultural practices and improve levels of crop yield. The development of GM crops has undoubtedly been a particularly controversial area of science over recent years, with many ethical, ecological and safety concerns being raised to debate the validity of their development (Conner *et al*, 2003). Nevertheless, the uptake of GM crops across the world has shown notable increase (Nap *et al*, 2003). A new prospect is the adventitious presence of GM material with medical or industrial applications through the advent of *Plant Molecular Pharming* (PMF), which has become a cause of heightened public concern (Dale, 2005). Consequently, government regulators require knowledge to assess and manage GM confinement in PMF crops (Hill, 2005). Therefore, sensible and scientifically legitimate exploration is necessary to appropriately develop prudent and forward-looking impact assessments of GM crop introduction and propagation.

The issue of containment is a key concern that needs to be explored when assessing risks involved with GM. There is a possibility that by introducing GM crops into a landscape, the genetic purity of conventional crops and wild plants will be altered as gene escape mechanisms cause GM genes to infiltrate these gene pools (Timmons *et al*, 1995). Gene transfer between plant populations may occur by seed dispersal and as the result of wind-mediated or pollinator-mediated cross-pollination (Ramsay *et al*, 2003), which makes problematic the implementation of containment, which is restriction to a threshold level outside a designated area. When measuring crop purity in landscapes that contain genetically modified plants, it is unrealistic to expect to find zero levels of GM seed in non-GM crops (Poppy & Wilkinson, 2005) and it is unenforceable in principle, because limits to detection necessarily exist. Instead, threshold tolerance levels provide a means to quantify acceptable levels of GM seed presence for a crop to be considered 'pure'. In Europe, EU Directives determine these thresholds (Nap *et* al, 2003) with current rules stating that non-GM food crops must contain no more than 0.9% of GM seed to be classified as non-GM, with crops harbouring levels beyond this labelled as Genetically Modified Organisms (CEC, 2003). This raises the question as to whether, and under what conditions, breaches of containment could be brought about through cross-pollination by bees.

In order to quantify likelihoods of gene transfer for a set of plant species, it is therefore imperative that pollen vectors for the plants are determined and their influence quantified. In

the case of animal pollen vectors, once the species is implicated in pollination, this will involve the analysis of movement patterns to determine how genetic transfer will be affected by pollinators picking up pollen in one plant population and depositing it on stigmas in another. As I have already discussed, animal movement patterns can be strongly influenced by foraging behaviour motivations. Therefore, an understanding of foraging behaviour could potentially enrich understanding of gene transfer levels for insect pollen vectors.

The scale across which GM gene escape can occur has been shown to be large, with gene transfers observed between populations separated at the kilometre-scale (Rieger *et al*, 2002; Ramsay *et al*, 2003). The low tolerance levels exemplified by the EU directives mean that sophisticated containment strategies may need to be considered to ensure threshold adherence in often patchy agricultural landscapes. This presents particular challenges in understanding, for example, influences of kilometre-scale field configurations on pollinator foraging movements and associated patterns of cross-pollination.

Here, I consider the specific case of oilseed rape (*Brassica napus L.*), which is a mass flowering crop that is widely grown in the UK, Europe and North America. *Brassica napus* has been shown to attract large densities of bumble bees (Westphal *et al*, 2003; Westphal *et al*, 2006). GM varieties of *B. napus* are already at the forefront of GM field trials within the UK (Ramsay *et al*, 2003). *B. napus* is both wind and insect pollinated (Hoyle *et al*, 2007) and it has been shown that whilst the predominant cause of gene flow in the species in many landscapes is wind (Hoyle *et al*, 2007), the influence of bumble bees as pollen vectors is non-trivial (Ramsay *et al*, 2003), particularly when bees are abundant (Hoyle *et al*, 2007). On average, abundances of key insect pollinators, such as bumble bees (*Bombus* spp.), are low in most winter-sown fields (the UK's commercial norm). However, surveys from the Farm Scale Evaluations (Department for Environment, Food and Rural Affairs, 2005) indicate that bumble bee densities exceed 0.03 bees m^{-2} in 5% of winter-sown fields, which implies that in these fields pollination by bumble bees accounts for ≥ 20% of the seed set. It is therefore important to include bumble bee pollination when assessing risks to GM containment.

1.3.3 How to determine the threat to GM containment posed by bee-mediated cross-pollination?

To understand the consequences of bumble bees foraging and pollinating within a landscape comprised of both GM and non-GM varieties of *B. napus*, we need to understand the landscape-

scale movements of bumble bees. Bumble bees are central place foragers that return to a nest when they have harvested nectar and pollen loads (Cresswell *et al*, 2000). It was traditionally assumed that bumble bees foraged close to their nests - within 50m - to minimise energy expenditure and maximise the benefits of resource intake (Dramstad, 1996). More recent evidence shows that bumble bees travel kilometre-scale distances from their nests in order to forage (Osborne *et al*, 1999; Dramstad, 1996; Saville *et al*, 1997; Knight *et al*, 2005; Greenleaf *et al*, 2007; Goulson and Stout, 2001; Darvill *et al*, 2004) and therefore their movement dynamics must be assessed over larger scales to determine their influence upon gene flow processes.

Whilst there have been many enlightening studies of bumble bee movements between patches of resource at smaller scales (e.g. Cresswell, 1997), quantifying kilometre-scale movements of individual bees has proven difficult. Bumble bees are typically very small and are therefore very difficult to observe in the field, and they also move fast, with speeds of 3 – 15ms^{-1} (Osborne *et al*, 1999). However, two methods have been developed to attempt to evaluate movements : mark-reobservation and harmonic radar. These are evaluated below.

Mark-reobservation methods involve the marking of discovered bumble bees in order to re-identify observed bees at a later stage (Saville *et al*, 1997). Bees are often marked with paints or tags such as discs attached to their bodies. Observers position themselves at potential visit sites for the bees, and they attempt to re-observe the marked bees to identify visited locations and establish a picture of movement patterns. The mark-reobservation method has intrinsic difficulties when applied to larger scales however (Dramstad, 1996), because as the number of sites that could be visited increases with area and thus with the square of the distance from the nest, there emerges a prohibitive increase in the observation effort required at more distant locations. Furthermore, it is notoriously difficult to mark bees individually at their nests for tracking, because bumble bee nests are typically concealed to ensure the security of their homes (Suzuki *et al*, 2007; Dramstad, 1996). Overall, the mark-reobservation method proves largely infeasible for our purpose.

Harmonic Radar approaches to tracking bumble bee movements appear to offer greater potential for dealing with larger scale environments. The method involves bees being temporarily trapped as they leave a nest, then fitted with a small transponder that relays the position of the bee to a receiver located at a central point (Osborne *et al*, 1999). Flight paths of bees are tracked and plotted to determine flight direction and range, and foraging site choices can be

identified by cross-referencing the bees' tracks with maps of the area. However, the method encounters problems in providing detailed observations of movement at the landscape scale. Tracking information from the transponders is 'line-of-sight' and is lost when bees fly behind obstacles on the horizon (such as hedgerows or variations in terrain height). Critically, the technology can also only track movements within a radius of 700m, which is inadequate for kilometre-scale analysis because it was found that bees often flew beyond this observation range. For example, of the 30 outward tracks presented by Osborne *et al* (1999), 10% were obscured in some way, 20% had a finish point that did not correspond to a known forage site and almost half flew beyond the tracking horizon. Overall, approximately three quarters of the departing bees did not have an identifiable forage site selection.

In summary, currently available empirical methods are inadequate for establishing accurate movement data for individual bumble bees over larger scales. Since such data is crucial if we are to assess the impact of bumble bees as pollen vectors in crops such as B. napus, in the interim it seems we must turn our attention to theoretical approaches to provide preliminary answers and to establish principles for future empirical testing.

1.3.4 Predicting pollinator mediated gene flow theoretically : the E-Psi-b model

If movement patterns of bees were known, we additionally require a means of predicting pollinator-mediated gene flow levels if we are to assess potential GM gene escape risks. A model developed by Cresswell et al (2002) proposes a method of quantifying the level of pollinator-mediated gene flow between two plant populations. The two populations are denoted as the 'source' and the 'sink'. The source population is the source of pollen and, in consequence, gene flow. In the case of the GM containment problem, the source population represents a population of GM plants. The sink population is any population to which gene flow from the source is possible. For this problem, the sink represents any conventional (non-GM) population whose seed purity we wish to ascertain. Specifically, I here define the level of gene flow as the *proportion* of seeds produced in the sink population whose paternity originates by cross-pollination from the source population.

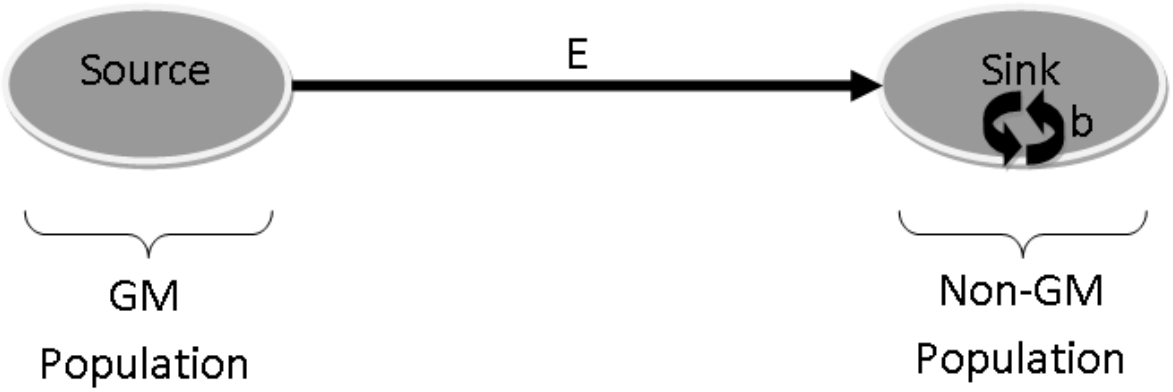

Figure 1.1. Key components of the E-Psi-b model, which predicts the level of gene flow from a 'Source' population into a 'Sink' population. E represents the proportion of pollinators arriving at the Sink from the Source, whilst b represents the number of flowers visited in the Sink. The Source and Sink populations may be considered to be GM and non-GM populations respectively, for the case of the GM containment problem.

The model makes two key assumptions. First, gene flow is proportional to *E*, the fraction of pollinators that arrive at a sink population directly from the source population. Second, gene flow is inversely proportional to *b*, the number of flowers consecutively visited by the pollinator in the sink after arriving from the source. As pollinators visit flowers within the sink population, source pollen they carry is exhausted and replaced with the sink's own pollen, therefore spreading sink pollen in gradually increased amounts. If a pollinator were to only visit a few flowers within a population, the proportion of seed set would show a high proportion of foreign (source-origin) pollen. Also, as pollinators spend longer in a sink population, foreign pollen is gradually lost through grooming and is brushed onto stigmas, and most subsequent pollinations are by pollen originating from within the source population itself.

The two assertions above form the basis of the predictive model :

$$\xi = \frac{E\psi}{b} \qquad \text{(Eq 1.1)}$$

where ξ is the proportion of the sink's seed where paternity originates from cross-pollination by source population flowers, E is the proportion of pollinators arriving at the sink from the source population, ψ is the number of fruits fully fertilised with the source population's pollen by each pollinator that arrives from the source population and b is the total number of fruits fertilised by each pollinator during a visit to the sink population. The value of ψ can be determined empirically and can be assumed to be fixed for any given pair of plant and pollinator

species. In the case of bumble bees foraging at large scales, *E* and *b* must be sought theoretically as they require precise knowledge of pollinator movements at landscape scales. To quantify *E*, we must have a means of predicting the plant populations that a bee will visit (in order to determine whether they are sink or source populations and the order in which they will be visited), and the relative frequency with which the bees move between the plant populations that they visit. To quantify *b* we must be able to predict how long bees will stay in the plant populations that they visit. But how can these predictions be made?

1.4 The assumption of optimality

1.4.1 An economics-based analysis of bumble bee movements

Analysis of behaviour becomes feasible if it has a single underlying cause that also can be mathematically described. One of the most predominant assumptions to be found throughout the field of animal behaviour study is the assumption that this cause is optimality – i.e. that the behaviour maximises the individual's (or genotype's) survival value. Natural selection dictates that evolutionary processes will promote the development of animals that are stronger in aspects of their life above other animals – the principle of the 'survival of the fittest' (Darwin, 1859). The need for the assumption arises because we cannot possibly know how an animal is making decisions, as we are unable to unravel the inner workings of their minds. We can build on the assumption as follows. Any given behaviour can be hypothesised as having a 'goal' – a reason why the behaviour is occurring – that increases survival value. The degree to which any given goal is being achieved can be assessed if we can identify a measurable proxy for survival value. Here, I will be using an economic currency (rate of resource uptake) as that proxy.

If we apply the assumption of optimality to foraging, we can see clearly how the assumption can be used. Foraging animals have finite resources both in terms of time and energy. In a simple case, the goal of a foraging animal may be either to collect resources as quickly as possible or to collect resources whilst expending as little energy as possible (Stephens and Krebs, 1986). Thus, the rate of harvesting resources is the proxy for survival. Operationally, to assess how closely these goals are being achieved, we can calculate the individual's Rate of Net Energetic Intake (RNEI). The RNEI quantifies the net energetic reward to the forager after considering the rewards obtained along with the costs incurred to obtain those rewards. Time-based RNEI calculation divides the total energy obtained by the time taken to do so whilst energy-based RNEI divides the total energy obtained by the total energy expended (Stephens and Krebs, 1986). The optimal behaviour of a foraging animal would be the behaviour that we would expect to be adopted given the assumption of optimality, and for the case I have outlined here

we would expect animals to behave in such a way as to make choices that maximise RNEI. RNEI is a suitable proxy for survival value in bumble bees, because the success of the annual bumble bee colony (i.e. production of queens and males) depends upon the rapid accumulation of resources within the timeframe of a single summer (Heinrich, 1979). The rate of resource uptake is therefore critical for bumble bee foraging and a credible currency for optimization as a proxy for fitness.

The assumption of optimality guides how we model the way in which behavioural processes work, as it determines the kind of decision processes an animal should make if it is trying to behave optimally (and adaptively). For example, when modelling an animal that is choosing between patches of food, we may struggle to determine how an animal should come to its decisions without the assumption of optimality. With the assumption in place however, we can form principles that characterise how the decision-making process should be modelled. In my patch choice example this may include principles such as "the animal should choose a patch with more food items available over a patch with fewer food items available" or "the animal should choose a patch that is closer over one that is further away". With such principles in place, a decision making rule can be formulated that determines how the model animal should behave in a given situation.

1.4.2 Optimal Foraging Theory

Optimal Foraging Theory (OFT) underpins a significant proportion of theoretical animal foraging research, and began with two landmark papers (Emlen, 1966; MacArthur and Pianka, 1966) that both independently outlined the need to model animal foraging behaviour by using principles from economics. By postulating that foraging animals attempt to maximise their gains whilst minimising their losses, these papers showed that by applying the goal of economic optimality to the foraging problem, theoretical models in economic terms could be constructed to identify the optimum cost-benefit strategy and thereby predict foraging behaviour. In effect, when applied to the foraging problem the assumption of optimality assumes that natural selection has maximised the rate of energetic intake for animals. In practice, gains are often considered as being energetic rewards from harvesting resources, whilst losses represent energy expenditure required to achieve these gains. An optimal foraging behaviour instigates the most efficient trade-off between these two considerations and usually maximises the rate of net energy intake (RNEI).

Applying OFT principles bypasses the need to comprehend exactly how an animal's mind works. Instead, modelled behaviours can be derived from first principles based upon how an animal that is behaving optimally *should* approach the foraging problem. OFT models can be expressed with a mathematical basis or using computational methods that can be applied to the modelling of any economically motivated forager whilst allowing for the nuances that differentiate different foraging animals. As such, theoretical models based upon OFT approach the scientific ideal of a general model (Mangel and Clark, 1986).

Early OFT models arguably were unrealistic, however. It was common for early OFT models to assume that foraging animals possessed 'complete information' about the state of their foraging environment (Pyke, 1984). For example, many models assumed that the animals knew the exact food availability in the patch in which they were foraging, as well as the availability of food elsewhere, even when foraging in patches whose scale was too large or ambiguous for their absolute quality of food supply to be determined, or when alternative locations had not been visited by the forager. The assumption of 'complete information' has been often criticised, but even later OFT models have often only partially relaxed the assumption. More realistic models that consider 'incomplete information' and that invoke learning by the forager to obtain such information exist, but are less common.

By building a model based on OFT principles, there is also an implicit assumption that the primary focus of the simulated animal is foraging. Many animals may have to consider other factors when they are foraging such as the avoidance of predators, the presence of competitors or seeking mates for reproduction. Pure OFT does not allow for these considerations, and whilst models can be built to incorporate such considerations, it is arguable that these behaviours affect foraging decisions at a lower level. It is therefore possible that the OFT approach may not always be an appropriate tool for forager modelling.

1.4.3 Possible criticisms of the assumption of optimality

The assumption of optimality is a fundamentally important assumption that forms the basis of many foraging behaviour theoretical studies. Indeed, the assumption is central to Optimal Foraging Theory (Schoener, 1971; Stephens & Krebs 1986), which is possibly the most significant and influential development in theoretical forager modelling. Invariably there have been criticisms of the assumption (e.g Pierce & Ollason, 1987). I now outline some of these criticisms and discuss their validity and relevance.

By assuming foraging performance is a heritable trait, it has been argued that the necessary implication is that foraging activities are separate from other behavioural activities in which the animal engages, because these traits are assumed to be 'optimised', which is problematic in the sense that their independence cannot be proven (Pierce and Ollason, 1987). Whilst it is reasonable to assert that there may be no individual genetic structures whose sole function is related to foraging activity, it is entirely reasonable to assume that there are genetic structures that at least partially control behaviours relating to foraging. I would deem this assumption reasonable as animals are often found to be efficient foragers with surprisingly rapid learning curves, suggesting some form of innate encoding towards the foraging problem. As long as there is some genetic basis that determines activities that influence foraging behaviour, then it would seem reasonable to assume selection for these actions, regardless of whether or not they are independent of other activities.

Additionally, even if there is overlap between genetic structures that result in foraging activity, this does not mean that the aim of a forager's behaviour is affected. For example, a forager employing predator-avoidance strategies could still be seen to be employing optimality in its foraging decisions, as long as the optimality is contextualised against the boundaries of possible optimality in that given scenario (Werner & Mittelbach, 1981). A forager that cannot forage in the most rewarding patch of resources because of the presence of predators there could still be seen to behave optimally if it were to choose the most rewarding patch that does not contain a threat of predation. Furthermore, the implication of the criticism is that if something cannot be proven then the assumption is necessarily invalid. Science as a discipline must acknowledge that gaps in its knowledge by making assumptions, as long as these assumptions are underpinned with current knowledge.

It has been argued that the assumption of optimality implies that foraging performance has been maximised because natural selection has been 'trying to solve a problem', but evolutionary processes have no foresight and are incremental (Pierce and Ollason, 1987). It is certainly true to say that evolution cannot possess the foresight to create optimal foragers by design, but this does not mean that functions that are relevant to the foraging problem cannot be incrementally optimised. It would seem reasonable to consider the need for all animals to obtain resources to be a universal constant, which would have been in place throughout the evolutionary development of the organism. As such, it is reasonable to theorise that natural selection could maximise the efficiency of functions that are necessary for resource gathering. Such promotions

would need no foresight, but would implicitly respond to relevant needs. Animals that are poor resource gatherers would not survive and would therefore not pass on their genes.

It has also been theorised that optimal strategies may not yet occur in nature as animals may not have sufficiently evolved to achieve optimality or may never be able to achieve optimality (Pierce and Ollason, 1987). This criticism is rendered largely irrelevant for many optimality-based theoretical models which acknowledge that the behaviours being simulated approximate real animal behaviour, and that it is likely that many organisms are not truly optimal in their decisions. However, the assumption of optimality guides how we can model behavioural decisions using a simple proxy for survival value, such as an economic currency. Provided that we are confident that foraging animals are at least tending towards optimality, and that the currency is likely to have predominant importance, we can determine the type of behavioural decisions that such an animal would make. Optimal foraging theory makes this step feasible.

Overall, these caveats are least worrisome when looking at the specific problem of bumble bees. It is generally considered that the threat of predation for bumble bees – crab spiders that sit inside flowers (Morse, 1986) - is often negligible. Consequently, it is unlikely that bumble bee foraging is affected to any significant degree by the possibility of predation. Worker bumble bees, which are by far the most assiduous flower visitors and, therefore, the most important pollinators, are also food-focused in the sense that, as energy maximisers, their primary motivation appears to be the gathering of resources for fuel and to feed their young (Best and Bierzychudek, 1982); as a sterile caste (workers lay eggs only when the queen is weak or dead), their behaviour is also unaffected by the distractions of mating.

Indeed, studies of bumble bee behaviour have often shown that when presented with a learning problem, bumble bees are remarkably efficient at solving the problem, and approach optimality. Such problems may include the differentiation between flowers based upon appearance, reward quantity or reward yield frequency (Keasar *et al*, 2002; Dukas and Real, 1993), and bumble bees often learn to both differentiate and update their perceptions when a change is made. Bumble bees are exemplary candidates for OFT.

Given that OFT is a significant field with over 40 years of development behind it, and given that bumble bees appear to be excellent candidates for optimal forager modelling, why can we not use existing OFT approaches to solve the problem I outlined earlier? In the following sections I

outline some pivotal existing approaches within the field of OFT and show why they are unsuitable to solve this problem.

1.4.4 The foraging 'problem' for bumble bees

Optimal foraging theory is typically applicable to three categories of foraging problem. The *patch departure problem* asks under what conditions an animal should leave a patch of resources in which it is currently foraging (Cartar and Abrahams, 1996; Pyke, 1984). A forager may decide to leave a patch based on the time since it last found a reward (Ydenberg, 1984), or the amount of resource it is currently harvesting from the patch (Charnov, 1976), amongst other triggers. The *patch choice problem* is focused on the how foragers decide upon the patches of resource in which they will forage (Pyke, 1984). I have shown above how foragers may decide between alternative patches. Most foraging animals will be faced with patch departure and patch choice problems, and so these problem classes are particularly significant. Finally, the *diet choice problem* asks which food types a foraging animal should select to eat (Stephens and Krebs, 1986; Pyke, 1984). These decisions may be based upon perceived quality of the food items, their availability, or their familiarity (Todd, 2000), amongst other factors. Each category of foraging problem has its own types of decisions that can be made, and the assumption of optimality can determine how we should model each of these, allowing these problems to be tackled using theoretical approaches. Here, I assume that bees are foraging in a landscape with a single, most-rewarding flower type (thus obviating the diet choice problem), and I therefore focus on the patch departure and patch choice problems, because they describe the way in which bees move among patches in such a landscape. In order to defend my choice of methodology, I now show why these specific problems cannot be addressed by a variety of other existing approaches.

1.5 Why we cannot use existing approaches

1.5.1 Why we cannot use the 'Ideal Free Distribution' theory

The Ideal Free Distribution (IFD) represents an equilibrium state that predicts that animals will distribute themselves across a foraging environment in such a way that no animal can improve its rate of energetic intake by moving away from its current location (Fretwell and Lucas, 1969; Bernstein *et al*, 1988). IFD predicts that patches containing more resources will be harvested by more individuals, with animals matching the proportions of food availability, so that the proportion of animals that forage in a given patch is equal to the proportion of food availability that the patch offers within the foraging environment as a whole. A necessary implication of

this is that all animals foraging within the environment will therefore experience the same rate of energetic gain (Bernstein *et al*, 1988). Given a set of parameters to define a group of foragers and a foraging environment, the equilibrium dispersion of individuals required for an Ideal Free Distribution can be determined mathematically.

Whilst there is some empirical evidence suggesting bumble bees may approximate Ideal Free Distributions in the wild (Dreisig, 1995; Cartar, 1991), the evidence relates to patches of flowers smaller in scale than that proposed by our problem. However, the most important limitation of the IFD approach is that it does not allow us to predict what happens before an equilibrium state or what would happen if the equilibrium state were to be disturbed; the patch-to-patch transitions made by foragers to establish the IFD equilibrium are not predicted, because IFD allows for the prediction of a distribution of foragers within a given scenario, but it does not predict how these distributions emerge. Thus, IFD is not useful to us, because to solve the bee-mediated gene flow problem we need to be able to predict the how frequently bees move between sets of fields to understand transition effects (i.e. to determine E), and how long they stay in those fields to understand dilution effects in gene flow (i.e. to determine b). Without these predictions we cannot use the E-Psi-b model to predict the level of bumble bee mediated gene flow between GM and non-GM populations. Therefore we cannot use the Ideal Free Distribution as a theoretical means of solving our problem.

1.5.2 Why we cannot use the Marginal Value Theorem

The Marginal Value Theorem (MVT) is a mathematical model that predicts departures of foraging animals from foraging patches that are distributed across a landscape (Charnov, 1976). MVT implements the assumption of optimality by specifying that the modelled forager adopts the behaviour that maximises its RNEI. The theorem states that if a forager is to behave optimally it should leave a patch once the RNEI for the current patch is less than the (marginal) average RNEI for the foraging environment as a whole. Assuming that the forager depletes resources within a patch, the initial rate of intake in a patch will be highest initially, but will decline as resources are harvested by the forager. A forager that leaves a patch too early will be sub-optimal because it will lose the benefit of the patch's abundance of resources, and spend too long on inter-patch travel, whilst one that leaves too late will be sub-optimal as it will miss the potential gains of higher RNEI in another patch.

The critical limitation for our intended application is that the MVT is typical of the classic OFT models in its reliance on the assumption of 'complete information'. In this case, the assumption is that the foragers know the absolute rate of intake for the patch in which they are currently foraging and also the average rate for the habitat as a whole. The assumption that foragers will know the exact average rate of gain across all patches is difficult to justify. For the MVT to work, the assumption must hold regardless of the number of patches a forager has visited, otherwise the question must be raised as to how decisions are being made outside of the MVT. If a forager is naive, and has visited few patches, then the assumption is invalid because a forager cannot possibly make a judgement about patches it has not yet visited. Bumble bee colonies contain naive bees due to relatively short worker lifespans (Brian, 1952).

The use of the MVT becomes even more problematic in spatially explicit models, because it specifies patch departure events but not patch choice. In our application, the patches chosen by a bee are fundamentally important, as the sequence of visits between patches of GM and non-GM type will dictate the potential for GM to non-GM gene flow events. MVT cannot therefore generate predictions that allow us to evaluate E within the E-Psi-b model, and so we must reject the theorem as a candidate tool for solving our problem.

1.5.3 Why we cannot use the Threshold Departure Rule (and other 'rules of thumb')

The Threshold Departure Rule (TDR) is a patch departure model that was formulated for bumble bees foraging at the plant-scale (Hodges, 1985). It predicts when a bee will leave a plant and when it will move onto another flower on the same plant. The theory is comprised of three rules. The first rule states that a bumble bee, upon its arrival at a plant, is equally likely to choose any of the flowers available. The second rule states a bee, having sampled a flower on a plant, will visit another flower on the same plant if the amount of nectar in the flower it just sampled exceeds a certain threshold; otherwise the bee will leave the plant. The third rule states that within-plant movement probabilities are constant and independent of experiences in prior plants, and that bees never revisit flowers. Whilst the Threshold Departure Rule in its literal form predicts plant-scale foraging dynamics that are at a much smaller scale than that outlined by our problem, I assume for the purposes of this assessment that the theory could be extrapolated to the larger scale, so that large patches of flowers are analogous to flowers of the original formulation. I therefore henceforth refer to plants as patches.

'Rules of thumb' like the TDR are attractive because they are biologically plausible, as they do not invoke complex cognitive processes and they have been shown to be very efficient approaches to problem solving (Todd, 2000). It is entirely plausible that bumble bees could implement a simple threshold decision mechanism. Moreover, it has been shown that the TDR simulates bee behaviour realistically and is energetically optimal (Hodges, 1985a). The TDR's strength is based on its biologically plausible simple rule of thumb that asks a bee to accept or reject based upon a single sample. No memory or learning mechanisms are required. At the scale of larger patches, this rule of thumb is insufficient to make informed patch departure decisions, however. In particular, as with the MVT, the TDR does not identify patch choices, only patch departures.

1.5.4 Why we cannot use Job Search theory

Job Search Theory (JST) derives from microeconomics and outlines a method for predicting the job that a job hunter will select (Hoyle and Cresswell, 2007). It assumes that the job hunter, or 'searcher', receives an infinite series of job offers sequentially and must decide whether to accept the current offer or wait for a potentially better offer in terms of salary. Job Search Theory predicts that an optimal searcher will only accept a job offer if it is judged to exceed a certain threshold (for example a minimum desired salary). The theory has clear parallel with foraging theory as a forager is essentially attempting to find the best 'offer' of a foraging site. Job Search Theory has therefore been applied to the foraging scenario (Hoyle and Cresswell, 2007). The model states that a forager encounters a patch and samples the food availability within. If the RNEI exceeds a given threshold, the forager will stay in the patch to forage until it is full to capacity, otherwise it will move to the next patch. The optimality problem is to choose the threshold for accepting an offer, or the foraging rate required to settle in a patch.

For our intended application, the critical shortcoming with JST's model is that it is formulated so that once a patch is selected by a forager, it continues to forage there until it is full to capacity. This constraint restricts the simulation of adaptive behaviours that respond to changes in the sampling experience of the patch. Also, unless a forager finds a patch that exceeds its critical acceptance threshold, the job-search theory based model assumes that a forager will continue to search for an acceptable patch indefinitely.

More critically, like the Marginal Value Theorem, Job Search Theory does not predict the patches to which a forager should move during search. Rather, the assumption is that a linear

sequence of patches is encountered by a forager who has no ability to choose the next patch or, moreover, to return to a previously visited one. I have already discussed in the previous section that we require a patch choice element to a theoretical approach in order to quantitatively evaluate E in the E-Psi-b model. Whilst the framework of Job Search Theory could arguably be extended to accommodate a patch selection mechanism that specifies the next patch to be 'encountered' by a forager, the other drawbacks I have discussed in this section would suggest that the Job Search Theory approach is not well suited to our outlined problem.

1.5.5 Why we cannot use HOOFS

The Hierarchical Object Oriented Foraging Simulator (HOOFS) is an *in silico* model that generates both patch choice and patch departure decisions (Beecham and Farnsworth, 1998). It is an example of a class of models known as Individual Based Models (IBMs) in which individual foragers are individually represented and explicitly modelled, so that populations are modelled as collections of individuals rather than atomic entities (Grimm and Railsback, 2005). This approach has the potential to create ecologically plausible models, not only because of the increased resolution in captured information, but also because it allows for the simulation of both implicit and explicit interactions between individuals in the population, and the behaviours that consequently emerge (Grimm and Railsback, 2005). HOOFS is also spatially explicit as it takes as an input a map of a landscape containing patches of forage, the distances between them and their positions relative to one another. Each forager within HOOFS follows a cycle which involves selecting a patch in which to forage, making a decision as to how long to stay within the patch based on information of the patch's quality, and harvesting resources within the patch.

To avoid making the assumption of 'complete information', the model provides each forager with increased information about a patch's quality the closer it is to that patch. The justification for this approach is that patches that are further away from a forager have likely been those that have not been recently visited, and therefore the forager's memory of such sites may be more inaccurate. The approach therefore retains the implication that foragers are expected to know the quality of patches that they have not yet visited, and as such it is still inappropriate for naive foragers or large landscapes containing many potential foraging sites.

The approach used by the HOOFS model to provide foragers with information means that sampling and learning dynamics are not simulated. The authors of the model justify this by reasoning that such mechanisms would typically result in forager knowledge tending towards

omniscience when environments are unchanging, and would necessitate the assumption of large forager memory capacities in dynamic environments. This justification appears flawed, however. Why, for example, is a gradual tendency towards omniscience problematic? If a foraging landscape is unchanging, it is reasonable to assume that a learning forager, given enough time, would eventually grasp the landscape's properties. For our application, the exclusion of 'learning by sampling' from the HOOFS model is a critical shortcoming, because we know that bumble bees are efficient learners. We must therefore reject HOOFs as a viable means of solving our problem.

1.6 Learning

1.6.1 Towards a learning-based approach

I have spent considerable time in this chapter highlighting the limitations of the learning elements of many traditional OFT-based foraging models. But why is it so important to model learning dynamics? Here, I identify three core reasons why learning dynamics must be a fundamental component of any theoretical approach that is implemented to solve our problem.

First, environments are changeable, even in hierarchies of floral rewards (Heinrich 1979). Foragers themselves can induce changes in the environment as their foraging activities may directly influence the absolute levels of food available or may indirectly attract or repel other foragers, in turn changing the pattern of food availability. In the specific case of bumble bees, plants can come into and out of bloom within a forager's lifetime. For example, *B. napus* blooms for several weeks (Hoyle and Cresswell, 2007a) but is likely to come out of peak bloom within the approximate three week lifespan of a typical worker bumble bee (Brian, 1952). Real environments are not static entities, and as such it is unreasonable to assume that foragers within these environments do not have the capacity to modify their responses accordingly. If the landscape changes, so too must the foraging policy.

Second, forager naivety necessitates learning if foraging is to become efficient. Whilst the assumption of optimality assumes that efficient foraging traits are passed on genetically, this can be feasibly only in a general capacity. Information specific to a particular foraging environment is unlikely to be passed on due to the changing nature of those environments. The only way to overcome naivety is by learning, whether this be guided by other members of the animal group or by self-initiation. With a worker bumble bee lifespan of only three weeks, set against a colony lifespan of 3 – 4 months, a significant proportion of the foraging force at any

given time will be considered naive. Naivety is therefore not a trivial consideration and learning dynamics must be included in any realistic model.

Third, the process of learning will alter a forager's behaviour so as to deviate from the strictly optimal foraging policy of an omniscient forager. This will affect movement dynamics and therefore affect both E and b of the E-Psi-b model. Learning will typically increase exploration as foragers must sample alternative foraging sites to ascertain their qualities and make informed comparisons and choices. If exploration levels increase, the levels of inter-patch traffic within a landscape will also increase. Assuming a constant resource capacity and a requirement to fill to capacity, the amount of time that foragers spend in patches on average may be lower when there are increased levels of inter-patch traffic, with foragers favouring exploration over exploitation. It is therefore imperative that learning processes are captured in our theoretical approaches to ensure increased realism in our gene flow predictions.

1.6.2 Learning and memory in bees

Bumble bees are very responsive to changes in reward, both in terms of reward volumes and variance (Pleasants, 1981; Ott *et al*, 1985; Cartar, 1991). This responsiveness to changes suggests that bees not only possess the capability to learn resource availabilities, but that they are very efficient learners that can quickly adapt to changes in their foraging environment (Dall *et al*, 2005). Their exhibition of risk aversion strategies (Harder and Real, 1987; Cartar, 1991) further demonstrates their sensitivity to reward information, and also implies that they possess an awareness of longer-term dynamics. Any theoretical model of bumble bee behaviour should acknowledge these traits by implementing efficient learning mechanisms.

N-armed bandit problems are a classic paradigm for evaluating learning algorithms. They outline a scenario in which there are multiple 'arms' that can be activated, and each of which offers a return (Keasar *et al*, 2002). Typically, arms differ in their rates of return. The optimal policy for the problem is to find the arm that offers the highest rate of return and to keep using it. Arms constitute foraging options, and can be analogous to patch choices. As such, N-armed bandit configurations have been empirically applied to bumble bees (Keasar *et al*, 2002). Two 'arms' are presented to the bees in the form of two different types of flower that differ in colour and probability of issuing a nectar reward when sampled. It was found that bees showed a gradual increasing preference for the more rewarding flower type, offering strong support for the theory that bees are optimal learners, in the sense that they adjust behaviour based on

experience to increase the rewards obtained. It was also found that bees showed greater fidelity to the most rewarding flower type when the difference between the rewards offered by the flowers was increased, suggesting that bees also adapt their behaviour according to the difficulty of the problem they face.

Learning requires memory capabilities, because learning must take into account prior experiences to develop future foraging behaviours. Bees have significant memory capabilities and appear to consider recent experiences when making flower choice decisions (Dukas and Real, 1993; Keasar *et al*, 2002). The harmonic radar study I discussed earlier within this chapter showed that bees often congregated on fields of flowers, with direct flight paths to such sites (Osborne *et al*, 1999), suggesting that bees maintain a spatial memory when foraging. Indeed, bees tend to visit the same foraging sites time after time (Osborne and Williams, 2001), and this site fidelity behaviour would necessitate both a spatial memory capacity so that they could remember where the site was, along with a capacity to recall the quality of the site from previous experience to judge whether a return is worthwhile. Bumble bees have also been observed to follow traplines (Thomson, 1996; Ohashi *et* al, 2008; Williams and Thomson, 1998) which indicates the existence of memory capabilities and suggests long term memory retention.

It therefore would seem that bumble bees are capable and efficient learners that can adapt to changing conditions in both space and time. It is therefore appropriate that models of bumble bee behaviour incorporate learning so as to have the potential to be realistic predictors. I therefore next explore models of foraging that incorporate learning.

1.7 Existing learning approaches

1.7.1 Why we cannot use an existing learning-based IBM approach

Bernstein et al (1988) developed an IBM that attempted to predict how populations of foragers disperse as a result of the foraging choices of individuals within the population. A forager selects a patch at random and continues to harvest the resources within the patch until it decides that it should leave for another patch. The amount of resources that a forager harvests from a given patch is determined based on the number of competing foragers present, how much they are interfering with the forager's activity, the amount of resource available, the time available to search for resources within the patch and the time taken to handle each resource item (which incorporates pursuit times for mobile prey). Foragers leave a patch when the rate of RNEI falls below their estimate for the average RNEI within the landscape, which they learn. This model

incorporates both an IBM framework and learning mechanisms. So is it suitable for tackling our problem?

Foragers learn the profitability of the environment as a whole, which means that the model does not have to rely upon the misguided assumption of complete information for this aspect of their decision-making. However, like the other models, Bernstein *et al*'s incorporates a random patch selection mechanism to determine the patch that should next be visited by a forager. This mechanism is entirely stochastic and therefore is not a significant development. The model focuses on patch departure, and does not properly address patch choice as a mechanism that could affect foraging behaviour and movement dynamics. Additionally, foragers do not learn individual patch qualities, so they cannot respond to spatial variation within the foraging environment.

1.7.2 Why we cannot use the Bayesian approach

Bayesian theory is a branch of probability theory that calculates conditional probabilities (McNamara *et al*, 2006) – the probability of something occurring given that something else has occurred. A Prior probability is a probability that is formed before experience with the events; a pre-conceived perception of the probability. Prior probabilities are altered with experience to create Posterior probabilities. When applied to the foraging problem, probabilities determine the likelihood of a given patch being a certain type – for example a patch type that is full of resources. Prior probabilities can be assumed to have been formed by evolutionary processes and represent the innate pre-conceived default perceptions of patches, when the forager is naive. A foraging animal samples resources within a patch, and learns by updating its estimates of the probability of a patch being of a certain type. Foragers leave patches based on information they are currently receiving, along with previous experience which has formed posterior probabilities.

A difficulty with the Bayesian approach is the problem of 'restricted world view' (McNamara *et al*, 2006). A Bayesian forager can never learn about something of which it has no innate knowledge. This is because the prior probability of such occurrences would be effectively zero, and so no transformational update of the prior probability can ever generate a non-zero posterior probability. From a theoretical stance this is problematic in the sense that all possible experiences must be considered in order to formulate a representative model. From an

ecological stance the limitation is unrealistic as it is unlikely that real animals would be completely non-adaptive when faced with a new experience.

Bayesian foraging principles would seem to be a sensible approach to modelling learning foragers. It acknowledges the individuality of foraging animals and provides their simulated counterparts with a capacity to learn based on their experience. Patch choice mechanisms can be easily included by implementing a simple rule of thumb, such as a comparative heuristic that states 'select the patch with the highest probability of being 100% rewarding'. Patch departure mechanisms can operate on a threshold-based scheme that instigates a departure when a probability for the current patch falls below a threshold level. In its raw form, however, the Bayesian approach demands the use of probability distributions which need to be updated as the forager learns. Updating probability distributions can become unwieldy and this may result in computational complexities as well as a diminished sense of ecological credibility, as animals are implicitly assumed to make complex calculations that are far beyond their neural capabilities. Clearly, the core ideas behind the Bayesian approach are promising, but a more satisfactory and credible implementation is required to harness these ideas when applied to the domain of animal foraging.

1.7.3 A promising solution - Reinforcement Learning

Reinforcement Learning (RL) is a machine learning technique in which a learning *agent* attempts to move towards a general *goal* via trial-and-error learning mechanisms, adapting behaviour based upon *feedback* received which indicates the success at reaching the goal (Sutton and Barto, 1999). Each agent is an individual with an independent learning history. The goal may be achieved by interacting with an *environment* which provides *rewards* to the agent when an interaction, or *action*, is performed. Depending on the magnitude of the reward received by the agent, the agent will accordingly adapt its behaviour in the future. Actions that yield greater rewards will reinforce the suitability of choosing the action that led to the reward, whilst less rewarding results have the opposite effect.

The applicability of RL as an approach to the foraging problem is clear, and a common implementation of the RL approach is known as an Artificial Neural Network (ANN). Essentially an ANN is based on processes derived from real neural architectures which are typically simplified. An ANN simulates an architecture consisting of artificial neurons and the strengths of the connections between them. A neuron can be thought of as instigating an action,

and the strengths of the connections to the neurons can be updated using RL to reinforce the instigation of more rewarding actions.

I here discuss an ANN and RL-based simulated bee that was designed to analyse the utility of the RL approach for theoretical bumble bee foraging behaviour analysis (Niv *et al*, 2002). The ANN used by the simulated bee was based on real neural architectures discovered in bumble bees. The model bee forages within a three-dimensional arena containing blue and yellow squares that represent two different types of flower. The flowers differ in their reward. The model bee advances towards the grid of squares one time unit at a time. At each time unit, the bee decides whether to continue on its current heading or to change direction, and this decision is based on the percentage of colour (a given flower type) that it can see within its field of view. Once the bee has landed on a 'flower', the bee consumes the nectar that the flower bore, updates its neural network according to the feedback it receives from this reward sample, and starts the process again. Genetic algorithms are used to evolve the learning rules within the ANN to promote those that are more effective and efficient.

It was found that, like the empirical studies of real bee behaviour outlined earlier, the model bee exhibited rapid learning by quickly showing increased preference for the more rewarding flower type, and quickly switching its preferences when the rewards of the flower types were switched. Further, the simulated bee, like real bees, exhibited probability-matching behaviour in which the proportion of visits made to a flower type closely approximated the probability of obtaining a reward from that flower type. It was also found that complex foraging behaviours were exhibited by the bee, emerging from only simple decision-making rules that are biologically tractable.

This study and its results demonstrate the alluring capability of RL-based approaches to produce and explain complex emergent foraging behaviours observed in optimal foragers such as bumble bees. The RL approach is both explicitly and implicitly biologically tractable; explicit in the sense that the implemented ANN frameworks were based on real bee neural architectures and therefore have a sense of direct validation, and implicit in the sense that the RL principle of positive reinforcement leading to increased choice of the corresponding action is directly analogous to the theoretical Optimal Forager in OFT. It is apparent that the RL approach yields high performance in the context of efficient foraging strategies and rapid learning techniques that I have shown to be necessary incorporations when modelling bumble bees. Whilst the specific case here is focused on a simple flower choice scenario, the strong results suggest that

the approach is ripe for extrapolation, potentially to the landscape-scale of our gene flow problem.

In its current form however, there are difficulties. Neural Networks have an inherent structural complexity that could lead to significant computational problems when attempting to solve larger scale and more complex scenarios. Additionally, even though neural networks are derived from real neural processes, the enormous complexity of real biological neural networks means that ANNs are necessarily significantly simplified representations of these real systems. As such, ANNs become 'black boxes' from which it becomes difficult to derive any biological meaning in terms of the processes that lead to the emergent behaviours. Whilst many theoretical studies highlight their lack of intention to provide philosophical insight into reasons why behaviours emerge (indeed, I too offer such a disclaimer in my own study), the lack of ability to extract any rationale behind the emergence of behaviours would seem to limit the potential scope of the approach. Consequently, whilst the Reinforcement Learning approach seems to be an ideally suited, proven and powerful means for tackling our problem, the Artificial Neural Network implementation of RL principles may not be ideal.

1.8 Summary

Animal foraging theory is a subset of animal behavioural study that is a significant field in its own right. The necessity for all animals to obtain resources for survival demonstrates that, along with the recognition that foraging processes can impact upon other ecological systems, there is a requirement for the scientific exploration of foraging behaviour. Foraging behaviours can be studied either empirically by directly observing the target animals, or theoretically by modelling them in some way.

A specific application of foraging behaviour study involves assessing the influence of bumble bees as a pollen vector when investigating the likelihood of GM to non-GM gene escape at the landscape-scale. The potentially significant foraging force of bumble bees when they are foraging on mass-flowering crops such as *B. napus* mean that such investigations are vital. We can use the E-Psi-b model to predict levels of pollinator mediated gene flow, but to parameterise the model we must know how often bees move between GM and non-GM plant populations and how long they stay within non-GM populations. Unfortunately, technological and other practical constraints severely hamper the ability to analyse bumble bee movements at the

landscape-scale. As such, we must turn to theoretical approaches to provide the answers we need.

A central assumption when approaching foraging behaviour theoretically is that animals are evolving towards being optimal foragers in the sense that they make decisions that approximate that of a theoretical ideal forager. Whilst the assumption has been criticised, the field of Optimal Foraging Theory has made significant developments for theoretical animal behaviour study and remains a powerful tool for predicting forager behaviours. However, traditional OFT approaches are unsuitable candidates to solve our problem because the additional assumptions that they make are unreasonable when applied to bumble bee foraging scenarios. In particular, the lack of simulation of learning mechanisms proves problematic when representing a forager that has been shown to exhibit strong learning capabilities, to forage in changing environments and whose populations contain non-trivial proportions of naive foragers.

Learning-based theoretical approaches have typically either only partially relaxed the assumption that foragers have complete information about the quality of patches within the environment, or have implemented structures that are too complex computationally and / or biologically. However, Reinforcement Learning principles have been shown to be a powerful approach to modelling bumble bee behaviour. Traditionally the implementation of Reinforcement Learning for ecological problems has been via Artificial Neural Networks that can be powerful but have computational and philosophical limitations. Acknowledging the potential of Reinforcement Learning as an approach, I outline in this thesis how principles of RL are combined with the ecologically credible Individual Based Modelling framework and the proven capabilities of a Linear Operator Learning Rule to formulate a modelling approach that overcomes many of the traditional criticisms of theoretical foraging studies, and which can be applied to the GM gene containment problem.

Chapter Two : HARVEST - An AI Foraging Model

2.1 Introduction

In Chapter One I presented a landscape-scale foraging problem relating to bumble bees, and drew attention to the necessity to understand bee movements at such scales. Given the limitations of empirical exploration at such scales, theoretical modelling is not only appropriate but necessary. I have designed and implemented a patch choice foraging model tailored for the central place foraging of bumble bees foraging within a landscape where resources are patchily distributed. My foraging model is based on the individual-based modelling (IBM) paradigm, which is a powerful modelling approach in ecology because of its ability to capture the dynamics of the individuals within a population.

My model is focused on learning processes which, as I discussed in the previous chapter, are vital ingredients for realistic foraging models. The process of learning a landscape of resources is likely to induce exploratory behaviours that will affect the patch-choice processes of foraging individuals, both in the shorter term as resource patch qualities are identified by the foragers, and also in the longer term with intricate dynamics between forager and landscape arising from local resource depletion activities that could affect the foraging behaviour of other foragers within the system. If learning is to be simulated then the conventional mathematical models of patch-choice behaviour are inappropriate as they must typically rely on complex probability distributions to determine outcomes (Rodriguez-Gironés and Vásquez, 1997; Mangel and Clark, 1986) or implement techniques such as Fuzzy Logic that require many additional parameters that can be difficult to estimate realistically (Inglis *et al*, 2001). Individual-based modelling allows for a more tractable approach to the problem; since individual foragers receive explicit representation, their individual learning mechanisms, histories and idiosyncratic internal states can be directly simulated. An individual forager is no longer simply a single probability within a distribution or an implicit sub-component of an equation, but instead is capable of individual decisions based on a potentially unique rule set that has been determined by learning from its own experience.

Specifically, I describe a model entitled HARVEST (**H**arvesting **A**nimal **R**einforced **V**alues and **EST**imates), which combines Reinforcement Learning (RL) with a Linear Operator learning rule. The model is built up from first principles of bumble bee behaviour, because bumble bees

are excellent candidates for representation as RL-based learning agents. I also outline some of the basic validation that was conducted to assure the integrity of the model.

2.2 Economically motivated foraging among patches

I define a patch as a discrete, uniquely identifiable and separate location bearing resources, having a set of clear boundaries that define its size and shape. That said, inter-patch distances may be arbitrarily small, because formally the definition of a single patch depends on the perceptive abilities of the forager (Kotliar and Wiens, 1990). Bumble bees are able to identify patches because they respond to patch boundaries (Plowright and Galen, 1985) and other linear features with edges, such as hedgerows (Cranmer, 2004).

In the model, each patch contains resources whose quantity, or value, is incompletely known to the forager (Emlen, 1966; Pyke, 1984; Rodriguez-Gironés and Vásquez, 1997; Kohlmann and Risenhoover, 1998; Keasar *et al*, 2002). The economically-motivated forager chooses patches that offer the most profitable rewards in terms of the quality, quantity or ease with which they can be obtained. An economically-motivated forager must also consider the spatial configuration of the resource patches presented, to minimise movement penalties and the associated opportunity costs. Furthermore, an economically-motivated forager requires a memory capacity that is sufficient to allow it to make a comparative judgement about the quality of the patch in which it has chosen to forage, and, if it is to be at all efficient, it should have the capacity to learn from experience and respond to changes in the environment. I now describe how this was achieved using the RL framework.

2.3 Reinforcement Learning

As a RL agent explores actions within the environment, those actions that bring the agent closer to achieving its goal are gradually reinforced and selected more frequently, whilst other actions are increasingly rejected (Sutton and Barto, 1999). In order for actions to be compared by the agent, their suitability is quantified by '*action-values*'. Every action available to the agent has an associated action-value, which quantifies the expected reward from pursuing that action. Action-values are the property of each agent and are based on its learning history. An agent selects from the possible actions whose range is based on the *state* of the system, defined both in terms of the agent's attributes and those attributes of the environment that the agent can observe. Selections are made in accordance with a (possibly stochastic) *policy* which

determines the actions that should be chosen, given the state of the system and the set of action-values.

The approach I have taken for the design of my model adopts the principles of RL but differs from the neural network-based and robotics-based RL implementations that are typical in ecological RL applications (Chapter One). Instead, I infuse the RL principles with an IBM framework to create an *in silico* approach that is not stymied by additional complexities in the foraging problem, and whose parameters can be directly traced back to the ecological scenarios being simulated and, therefore, take realistic and measurable values. I first explore the principles of bumble bee behaviour that I wanted to capture, which I discuss in the next section.

2.4 Formulating the first principles of bumble bee behaviour

Bumble bees are pollinators of many crops and wild flowers in the northern hemisphere. For example, *B. Napus* attracts significant bumble bee foraging activity as it is a widely grown crop in many parts of the world (Damgaard and Kjellsson, 2005), and provides abundant supplies of nectar and pollen (Pham-Delegue *et al*, 1993). Worker bees exclusively focus on the collection of resources (nectar and pollen) from flowers (Heinrich, 1983), and they are social in the sense that their foraging efforts aid the colony as a whole (Cartar and Dill, 1990). Unlike honeybees, however, experienced bumble bees forage largely by individual initiative (but see Chittka and Leadbeater, 2005) and do not possess a 'dance language' for socially directed foraging, as occurs in honey bees, *Apis mellifera* L. (Visscher and Seeley, 1982).

Bumble bees are 'central-place foragers' because they forage from a single nest within which the colony lives (Cresswell *et al*, 2000). During foraging activity, bumble bees leave the nest and return when they have collected sufficient nectar and pollen before flying out to forage again. Fields of mass flowering crops, such as *B. napus*, are particularly attractive to bumble bees (Cresswell, 1999; Westphal *et al*, 2003; Westphal *et al*, 2006), as they offer potentially plentiful amounts of nectar and pollen within a small foraging radius.

Bumble bees forage over larger landscape-scale distances of several kilometres (Osborne *et al*, 1999). Bees typically only take a short amount of time to sample a flower - usually around 3 seconds (Cresswell, 1999) and receive rewards from the nectar and pollen they harvest. Bumble bees respond to changes in nectar availability (Pleasants, 1981) and pollen availability

(Robertson *et al*, 1999; Rasheed and Harder, 1997), suggesting adaptability based on their sampling experiences.

Based upon these characteristics of bumble bee behaviour, I derived eight principles to shape the design of the model. First, bees will be focused on a single resource type in the landscape, which could be either nectar or pollen, or a combination of both. In *B. napus*, the gradual replenishment of nectar and pollen after a bee visits (Pierre *et al*, 1999; Koltowski, 2002) means that their levels are effectively correlated. Bees typically collect either nectar solely or nectar and pollen (Heinrich, 1976). In either case, the value of the flower is related to the length of time since it has been last visited. Second, bees should choose between explicit foraging locations representing patches of resource. Since the levels of resources (nectar and pollen) motivate foraging for bees (Heinrich, 1983; Rasheed and Harder, 1997), it would be reasonable to consider patches as clusters of resource units that vary in value (i.e. collections of flowers whose nectar and pollen levels range from zero to some upper limit).

Third, bees should always choose the action that they believe will lead to the most profitable foraging activity at any given time. Since real bees are highly economically motivated and averse to taking risks, this would seem a suitable assumption as to how bees approach the patch choice problem. Fourth, since we have accepted that bees are risk averse and we know that nectar is required in constant supply to act as a source of fuel, then as well as considering the quality of resource patches in the landscape in terms of food availability, bees should also consider the cost in energy and time of moving to the patches being considered. The cost associated with moving between patches is likely to be relatively trivial at the smaller patch scale, but at the landscape scale where movements take place over many kilometres, moving to another resource location will use up much more time and energy. Long-distance travel could result in significant decreases to foraging efficiency, or harvesting rate, if insufficient resources were discovered at the candidate location.

Fifth, the judgements made by bees as to the quality of a patch should be based on the sampling experiences they have collated. Since real bees respond to sampling experience (Pleasants, 1981; Robertson *et al*, 1999), this proviso is necessary to capture real bee learning dynamics. Sixth, bees must retain a memory of past foraging experiences in patches. This demand has to be met if we accept that bees undergo learning as they forage. I do not assume that bees must retain a memory of all of their past flower encounters in patches, however, as bees typically visit thousands of flowers (Michener, 1974) and such an assumption may therefore be biologically

implausible. Instead we merely require the model bees to retain a memory of their last evaluation of a patch's quality, which itself represents an accumulation of previous flower encounters.

Seventh, since real bees forage according to individual initiative, all foraging and learning experiences for bees should be independent of the foraging experiences of other bees. Obviously there is the potential for indirect inter-bee interaction via bee-induced changes to resource availabilities in the landscape, as the foraging impact of one bee taking resources from a patch may affect the foraging experiences of other bees visiting the same patch, which will be particularly relevant for scenarios where resources are scarce. However, this indirect interaction alone will represent the extent of inter-bee interactions. Eighth, since bees are social foragers because they forage for the benefit of the colony as a whole, they should return to the nest to deposit their nectar and pollen harvests once their capacity is reached.

The eight first principles outlined above form a design map for a model bee. It is important to reiterate that all of the above principles are derived from general behavioural traits that either have empirical validation or may be reasonably based on observations, rather than coming from specific quantitative results obtained from empirical findings in an attempt to create a virtual agent that will best fit these observations (i.e. there is no circularity here). Furthermore, the preceding principles are general, such that if the model is shown to be an accurate representation of bumble bee foraging activity, then it should therefore be applicable to a variety of foraging scenarios, in particular to those outside the scope of empirical tractability such as landscape-scale movement behaviours.

2.5 HARVEST

2.5.1 Description of the model

Each RL agent in the model represents a single bumble bee capable of learning. The bees are presented with a number of patches containing rewards, hereafter termed 'nectar', and must choose between them. The collection of bees foraging simultaneously in the simulation is referred to as the *colony*, whose size is determined by a parameter denoted *B*. The environment is defined as the *landscape* in which the colony is foraging. More specifically, the landscape is composed of *L* patches containing resources, a single nest from which all bees in the colony commence their foraging activity and to which they return to deposit collected nectar, and positive distances separating resource patches from each other and the nest. Each resource

patch in the landscape represents a collection of nectar-bearing flowers, with each patch containing at least one flower, and the number of flowers in patch *i* given by F_i. For the purposes of simplification, I assume that a patch is simply a collection of flowers. Whilst this is a simplification of some real landscapes where patches are made up of many individual plants, for simplicity I assume that resource patches are dense collections of plants with only trivial distances between flowers, compared to inter-patch distances. In summary, each patch is viewed as a collection of flowers rather than a collection of plants bearing flowers (in fact, the model is able to incorporate a definition of a patch as a single plant should such high resolution simulation be required).

I also assume that all bees are aware of the number of patches in the landscape and their locations from the start of the simulation. In effect, I assume that bees instantly acquire an aerial map of the landscape. This is not unreasonable, because bumble bees have been observed making circling manoeuvres or 'exploratory flights' which are likely for the purpose of seeking foraging sites (Manning, 1956) and soon show strong navigational abilities even when placed in unfamiliar surroundings (Goulson and Stout, 2001).

Rewards are provided to the bees in the form of nectar obtained from the flowers. When a bee samples a flower, it extracts all of the nectar being provided by the flower (Manning, 1956), obtaining a quantity of nectar *s(i)*. For simplicity, I implement a binary nectar reward system in which flowers are either full with nectar (*full flowers*) or they contain no nectar at all (*empty flowers*). The parameters R_{full} and R_{empty} define the quantity of nectar in full and empty flowers respectively. In reality nectar distributions are continuous (Cresswell, 1990), however I believe that this simplification is reasonable for two reasons. First, a binary reward system approximates the 'threshold' decision making that has been empirically demonstrated in bumble bees and codified as the *Threshold Departure Rule* (Hodges, 1985). Hodges (1985) found that bees appear to make small-scale movement decisions based upon whether or not a nectar reward is above or below a particular threshold. Thus, my binary reward implementation is consistent with this observed behavioural trait. Second, the implementation of continuous nectar distributions would require much more complex assumptions in the model, because individual flowers would require a replenishment model. Patterns of nectar production can be complex and I wish to avoid making the model 'parochial' by closely simulating one system. This philosophy of retaining simplicity attempts to promote an ease of understanding as to why behaviours exhibited by the model bees emerge.

Each bee can carry a pre-defined quantity of nectar *C*, which is known as the *fixed nectar carrying capacity*. This is analogous to the capacity of the bee's honey sac. The fixed nectar carrying capacity is constant for all bees in the system. Nectar quantities are universally expressed in *nectar units*, which are the 'atomic' units of nectar used in the model. In my binary nectar reward implementation, full flowers contain one unit of nectar ($R_{full} = 1$) whilst empty flowers contain zero units ($R_{empty} = 0$). As such, nectar harvests made by a bee can be understood to be the number of full flowers the bee has encountered. Since bees attempt to maximise their rate of energetic gain, the goal of each bee is to fill to capacity with nectar as quickly as possible. A HARVEST bee can only return to the nest once full to capacity with nectar. Whilst real bees may return to their nest with somewhat varying nectar loads (Roubik and Buchmann, 1984), I make this assumption for the purposes of simplification. Foraging activity in the model is split into *foraging bouts.* Every time a bee leaves the nest it begins a foraging bout which only ends once it has returned to the nest with a full load of nectar. Once the nectar has been deposited at the nest, the bee leaves the nest again and begins a new foraging bout. Again, for simplification, I assume the process of depositing nectar to be instantaneous, so that the only time penalties accrued are those when moving outside the nest.

By implementing foraging bouts we effectively define a time horizon (Krebs and Kacelnik, 1984) over which foraging efficiency should be maximised. Therefore, each bee attempts to maximise the rate at which it collects nectar within a foraging bout by attempting to fill to capacity as quickly as possible. The actions available to each bee to achieve this goal are either to move to another patch in the system or, as long as the bee is not currently at the nest, to stay at the current location. Once a bee selects an action, it flies to the destination patch and incurs a cost for travelling which is defined by the flight time between the patches. The energetic costs of flight are assumed to be negligible compared to the opportunity cost of flight time (Cresswell *et al*, 2000). If the action is to stay in the current patch then no travel cost is incurred. A bee foraging within a patch encounters flowers stochastically, so that the probability that the next flower is a full flower is given by the proportion of full flowers in the patch. Real bees may exhibit traplining or systematic foraging behaviour within patches of resource in some circumstances (Thomson, 1996; Ohashi *et al*, 2008; Williams and Thomson, 1998). However, when patches of flowers are very large, systematic foraging is least consequential. Therefore, my model based on random selection of flowers applies best to large patches such as agricultural fields of *B. napus*. I discuss the possibility and implications of traplining implementation more extensively in Chapter Seven.

Each patch in the landscape has an associated *patch quality* which quantifies the nectar availability in the patch, and I represent this as *Q(i)*, which is the actual quality of patch *i*. Specifically, patch qualities are expressed as the proportion of full flowers in the patch. For example a *Q(i)* value of 0.6 would indicate that 60% of the flowers within patch *i* were full and 40% were empty. This proportion-based expression of patch qualities allows patches of different sizes to be compared in terms of the expected foraging efficiency expected therein. Also, patch quality represents the mean reward expected from a patch. Thus, the probability of a bee being presented with a full or an empty flower is directly drawn from a Bernoulli distribution with the probability of success equal to the patch quality. *Q* defines the initial quality of patches in the landscape for those scenarios where the initial reward availability in the landscape is even and homogenous.

Since the model bees do not have complete information about the quality of patches in a landscape and must instead learn them, each bee maintains an *estimated patch quality* for each patch in the system. I express this as *q(i)*, which is the estimated quality of patch *i*. At the beginning of the simulation, a bee has a default estimate of the quality of the patches in the landscape, much like the concept of a Bayesian prior probability that I discussed in Chapter One. In this study, I typically assign a default initial estimate of $q = 0.5$ for all patches, to provide each bee with a default assumption of a moderately rich landscape that is neither plentiful nor barren. As bees select patches and encounter flowers therein, their estimated patch quality for the patch in which they are foraging changes. In general terms, when a bee encounters a full flower within a patch, its estimate of the quality of that patch increases. In RL terminology this is positive feedback from its foraging experience in that patch. Similarly, when a bee encounters an empty flower within a patch, its estimate of the patch's quality decreases.

More specifically, the approach I use to implement updates to estimated patch qualities is known as the Linear Operator Learning Rule (Beauchamp, 2000; Ward *et al*, 2000). Essentially, the Linear Operator Learning Rule has three components : an accumulative memory of the previous quantification assigned to an experience, the quantified value of the new experience, and a learning rate parameter which determines the level of influence of the new experience on the previously accumulated quantification. Formally, the Linear Operator Learning Rule I implement is expressed as the weighted sum of new and old experience :

$$q(i)_{new} = \beta s(i) + (1-\beta) q(i)_{old} \qquad \text{(Eq 2.1)}$$

where $q(i)_{new}$ is the bee's updated estimated patch quality for patch i, $q(i)_{old}$ is the prior estimate, $s(i)$ is the value of the reward obtained from the newest sample in patch i (expressed in nectar units), and β is the learning rate parameter. β may also be thought of as controlling the sensitivity of a bee to a new experience or, put another way, as determining the memory capabilities of the bee (i.e. the agent's memory capacity is weaker as β approaches unity).

β is constant for all bees in the colony and lies in the range $0 \leq \beta \leq 1$. It can be seen from equation 2.1 that higher values for β lead to bees showing extreme sensitivity to each new flower sample, and a reduced memory of past foraging experiences. At the maximum of $\beta = 1$, no consideration is given to previous experience in the patch, so in the binary nectar reward system patches are only ever estimated to be completely full with full flowers or have no nectar at all. Similarly, smaller values of β place more importance upon prior experience, with the minimum of $\beta = 0$ effectively turning off learning completely as bees disregard all new experiences. Importantly, this learning rule is biologically plausible as it uses no additional 'physical' memory regardless of the number of flowers that are encountered. Instead, an accumulative perception of experience in each patch is maintained.

The state of a bee at a given time is determined by two further factors : c - the *remaining nectar carrying capacity* of the bee - and its current location within the landscape. The state of a bee is used to formulate the action-value to be associated with each action in the model. Recall that action-values are the quantifications of the goal-appropriateness of choosing an action that moves the bee towards its goal. When considering whether to enter a patch, the bee must first assess its expectations of the foraging experience to be had at the candidate destination. Since the model's incompletely informed bees cannot use true patch qualities to determine this, they must use their estimated patch qualities maintained by equation 2.1. Based on the estimate of a patch's quality and the amount of nectar required to fill to capacity, the bee can calculate an expected time to fill to capacity if they were to continue foraging in the candidate destination patch until full to capacity. Thus, action-values take the units of time. In addition, bees must also consider the movement penalty in units of time associated with travelling to the patch. All distances in the model are expressed in *time units* where the atomic unit is the length of time needed to sample a single flower. Viewed in this way, movement penalties can be understood relative to the flower sampling sacrifice required, or opportunity cost. I neglect the energetic cost of flight movements in inter-patch travel, because bees often hover and fly whilst visiting

flowers within a patch (Manning, 1956), so the marginal energetic cost of inter-patch travel is small. Formally, we combine the considerations above into the action value calculation :

$$v(i, j) = \frac{c}{q(j)} h + t(i, j) \qquad \text{(Eq 2.2)}$$

where *v(i, j)* is the bee's action-value associated with choosing to forage in patch *j* given that it is currently situated at location *i*, *c* is the bee's remaining nectar carrying capacity, *q*(*j*) is the estimated patch quality for patch *j*, *h* is the handling time per blossom and *t*(*i*, *j*) is the time required to fly from patch *i* to patch *j* in time units (which is zero when *i* = *j*). Location *i* may represent either a patch or the bee's nest, but location *j* must always represent a patch, as returns to the nest are not valid actions; they are triggered only when a bee fills to capacity and cannot be chosen. Superior actions are identified as those with lower corresponding action-values, and the bee implements a simple "greedy" policy : to choose the action with the lowest action-value (breaking ties randomly). Patch quality updates, action-value updates and action choices are made after every flower encounter to enable the bees to respond quickly to changing circumstances or perceptions.

It should be noted that many Reinforcement Learning models implement a parameter which determines the probability that an agent will select an action that is not considered to be the optimal at any given action choice, in order to take an exploratory action (Sutton and Barto, 1999). However, my initial studies with one of the first implementations of the model showed that this parameter was not necessary because exploration is prompted intrinsically by the stochastic nature of nectar sampling (specifically, a bee will leave a patch after a 'run of bad luck', or successive empty flowers that have depressed the local patch's action value). This is interesting, because policies that have built-in exploration mechanisms are near-universal in the field of machine learning, but prove to be unnecessary, even detrimental, here. I provide further details of this in Appendix B.

When a bee samples a flower it extracts all of the nectar from the flower if any is present, turning a full flower into an empty flower. I call this the *depletion effect*. After a pre-defined interval has elapsed in the model, known as the *replenishment interval* R_i, an empty flower is replenished and changes from an empty flower into a full flower. I call this the *replenishment effect*, and this mirrors the nectar replenishment capabilities of real plants (Pierre *et al*, 1999).

Flowers that were empty at the start of the simulation are randomly assigned 'residual' replenishment intervals drawn from a range with a minimum of 1 and a maximum of R_i.

Each run of the simulation with a given set of parameter values is termed a *trial*, and multiple trials can be repeated with the same parameter set to obtain more accurate average results. Within this study I call such a group of trials a *trial set*. Each trial also has a time limit (specified in time units) that determines the length of foraging to be simulated before the simulation terminates. The *global clock* records the time that has elapsed since the simulation began, and is given in time units.

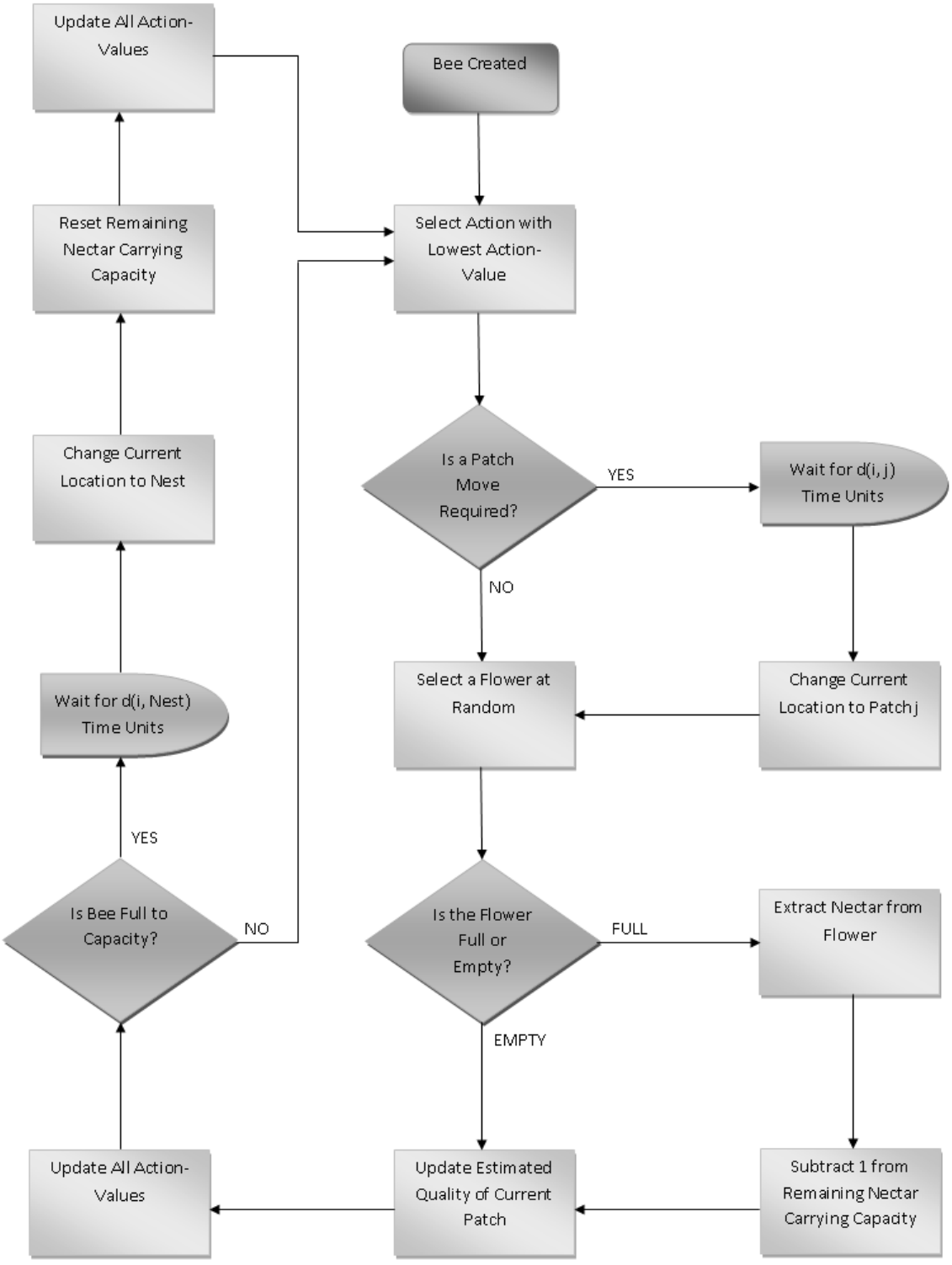

Figure 2.1. Flow Chart showing the foraging process of a bee within the HARVEST model. HARVEST bees learn using trial-and-error methods, and update their perceptions based on reward feedback.

Parameter	Identifier
ξ	Proportion of sink patch's seed with source patch paternity mediated by cross-pollination
E	Fraction of pollinators arriving at sink from source
Ψ	Number of fruits fully fertilised in sink with source pollen by each pollinator in a sink visit
b	Total sink fruits fertilised by each pollinator in a sink visit
B	Number of bees in the grid
L	Number of patches in landscape
Q	Initial quality of patches
Q(i)	Actual quality of patch i
q(i)	Estimated quality of patch i
β	Sensitivity to individual flower sample
s(i)	Value of the reward from latest sample in patch i
v(i, j)	Action-value for visiting patch j when currently in patch i
C	Bee's nectar carrying capacity
c	Bee's remaining nectar carrying capacity
h	Flower handling time
t(i, j)	Flight time between patches i and j
F_i	Number of flowers in patch i
I_i	Replenishment interval for patch i
P(x)	Bee's foraging efficiency in foraging bout x
G(x)	Elapsed time on the global clock since foraging bout x began
R_{full}	Reward (in nectar units) offered by a full flower
R_{empty}	Reward (in nectar units) offered by an empty flower

Table 2.1. Summary of parameters used by the model.

2.5.2 Measuring foraging performance

Foraging rate is a key component of my foraging model, and its maximisation is the goal of each bee. I use the term *performance* to describe the foraging rate of a bee in the simulation. Theoretically, the highest foraging rate would require a bee to move to the nearest patch from the nest, never encounter an empty flower, and never move to another patch. This would ensure that the bee filled to capacity as quickly as possible and represents an ideal realisation of its goal. Likewise, the lowest foraging rate would arise from encountering only empty flowers, the maximum number of inter-patch moves possible within the time, and the longest inter-patch moves. I therefore express a bee's performance in terms of the amount of nectar collected and the time spent actively foraging, which in my model equates to the proportion of time spent sampling full flowers within the total time spent foraging. I calculate performance on a per-bout basis for each bee. Formally, we define the foraging efficiency in bout x as :

$$P(x) = \frac{(C - c)}{G(x)} \qquad \text{(Eq 2.3)}$$

where C is the fixed nectar carrying capacity of the bee, c is the bee's remaining nectar capacity and $G(x)$ is the time elapsed on the global clock since the bee started foraging bout x. $P(x)$ is measured in nectar units per time units. A foraging bout is completed when $c = 0$. Equation 2.3 implements the conventional rate of net energetic intake (RNEI) calculation method (Stephens and Krebs, 1986).

$P(x)$ represents a proportion, and within the simulation it has a minimum of 0 and a maximum of 1. However, since my model assumes that all inter-patch and nest-to-patch distances in the model are non-zero, then the theoretical maximum can never be attained as no flower sampling can occur without transit. I am attempting to measure the foraging efficiency of incompletely informed learning foragers and it is useful to compare them to some form of achievable benchmark. Therefore, I compare the foraging performance of incompletely informed learning bees with otherwise equivalent but completely informed non-learning counterparts, or *omniscient bees*.

Omniscient bees follow exactly the same rules as learning bees, but unlike learners they are aware of the true patch qualities of all patches at all times; in other words $q(j)$ from equation 2.2

is replaced by *Q(j)*, which is the actual patch quality of patch *j*. They are similar to the foragers of many traditional OFT models, in which complete information is assumed. Omniscient bees must still encounter flowers within their chosen patch stochastically, however, since they only know which patches offer the best quality foraging, not whether the flower they next encounter will be full or empty. By testing learner bees against omniscient bees, we can express foraging performance in terms of the percentage of omniscient performance obtained, with higher performance percentages implying that the bees have more accurately learned which resource patches offer preferential foraging rates.

2.5.3 Calculating bee-mediated gene flow

In Chapter One I presented a model for predicting pollinator-mediated gene flow, which for this patch choice model we can re-write as :

$$\xi_{AB} = \frac{E_{AB}\Psi}{b} \qquad \text{(Eq 2.4)}$$

where ξ_{AB} is the proportion of pollen deposited in patch B that originates from patch A, E_{AB} is the proportion of pollinators that move directly from patch A to patch B, b is the number of flowers pollinated by the pollinator in patch B, and ψ is the number of fruits pollinated with patch A pollen by a pollinator arriving from patch B. ψ must be determined by empirical methods (Cresswell and Hoyle, 2006), but is reasonably assumed as constant for a given pollinator foraging on a given plant species. That leaves only E_{AB} and b values to be determined, which the model is able to generate.

After each trial in the model, a *transition matrix* is generated which shows us the proportions of action selections in terms of the total actions selected by the colony. Since actions equate to moves between patches or the action of staying within a patch, this provides us with details of inter-patch transitions. Each row in the matrix represents an origin patch and each column a destination patch, with the corresponding entry representing the proportion of moves made by the colony from origin to destination over the course of the trial. HARVEST outputs the transition matrix as a comma-separated values file. Appendix A offers a worked example of how this file is translated into the transition matrix described above and used to evaluate E_{AB}.

I use the transition matrix to populate E_{AB} by looking up the value for which the nominated source patch is the origin patch and the nominated sink patch is the destination patch. The model explicitly tracks the number of flowers visited by each bee in each patch over the course of a trial, and a mean residence per bee is obtained for each patch. The mean residence for the nominated sink patch therefore provides a value for b.

From the perspective of GM containment, the worst case gene-flow scenario occurs in equation 2.4 when $E = 1$ and $\psi = b$, as this implies that every flower in the sink population that is visited by a pollinator is fertilised to its maximum extent with source pollen. The best possible case simply involves no pollinators making a direct move from source to sink, so that $E_{AB} = 0$ and therefore $\xi_{AB} = 0$. For gene flow results between these extremes, we can see that we obtain an increase in predicted gene flow levels as sink residence is shortened ($b \rightarrow 0$) and the frequency of direct source to sink moves increases ($E_{AB} \rightarrow 1$). ψ has been determined empirically for bumble bees foraging on *B. napus*, and has been shown to be approximately 1 fruit (Cresswell *et al*, 2002). For simplicity, I use a ψ value of 1 to simulate this pairing of pollinator and plant species.

2.6 Validation of performance

As with any *in silico* implementation, validation of the simulation is required before the parameter space is explored more extensively. In this section I present details of some of the significant checks that were conducted, whereby basic tests were carried out to discover if predicted behaviours emerged from the model under sets of parameter values where the required behaviour was foreseeable.

A key feature of the model is the expression of patch qualities as the proportion of full flowers in the patch, since this determines the probability of a bee encountering a full flower in the stochastic binary reward flower system I have configured. I would therefore expect that, given a sufficient period of time, the bees foraging in a patch should encounter full flowers at a rate that closely approximates the quality of the patch. To test for this expected behaviour, I configured a series of trials in a 3-patch landscape with a single bee. Two of the patches were not analysed and were assigned extremely low patch qualities to deter the bee from spending any significant time within them, whilst the third patch was positioned closest to the nest and was provided with patch qualities that were always higher than those of the other two patches. Each patch contained a total of 10 flowers, and the analysed patch was tested with assigned

qualities between 0.1 and 1.0 inclusive. In addition, the bee was provided with accurate knowledge of the patch qualities from the start, and depletion effects were disabled. Each trial was terminated after the bee had conducted 1,000 foraging bouts.

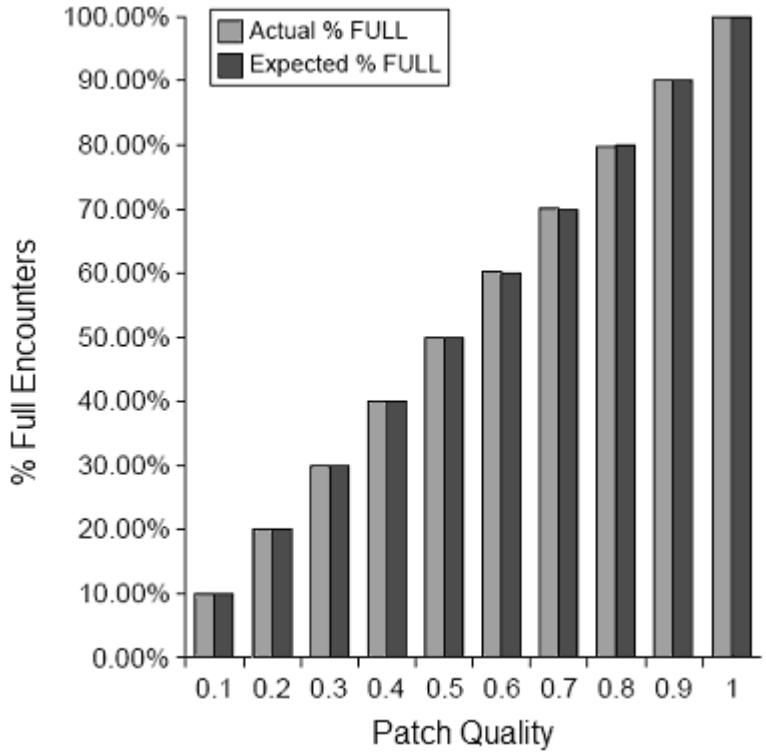

Figure 2.2 – Percentage of a bee's encounters with full flowers in a patch with varying patch qualities compared to the expected percentage of full flower encounters after 1000 foraging bouts of foraging activity. X-axis shows patch quality, whilst y-axis shows the percentage of encounters that bees made with full flowers. 3 patch landscape tested, with one patch analysed, and a single bee monitored over 1000 foraging bouts.

Figure 2.2 compares the expected percentage of flower encounters that would involve full flowers with the actual percentage experienced by the bee. The data sets are extremely close and imply that the implemented stochastic flower encounter mechanism works as intended.

When bees sample flowers in a patch, they alter their estimates of the patch's quality according to the type of flower they sample. Assuming $\beta > 0$, the sampling of a full flower should increase a bee's estimate of patch quality whilst the sampling of an empty flower should have the opposite effect. I ran trials to ensure that this fundamental adaptive behaviour mechanism

worked. A landscape of three foraging patches was configured; one patch (patch A) was configured as the closest patch to the nest whilst the other two patches were placed such that their distance to the nest was ten times that of patch A. Both of these distant patches were also provided with a patch quality of 0, and the single foraging bee was provided with complete patch quality information. This ensured that the bee focused its foraging activity on patch A from the start. The bee's estimated patch quality for patch A was analysed over the course of the trial and compared to the times at which empty flowers were sampled by the bee. I observed that every time an empty flower was encountered, the estimate of the patch's quality decreased whilst full flower encounters caused increases in the estimate. At those times when the bee was moving between patches, or moving to or from the nest, there were no changes in patch quality estimates, as the bee was not sampling flowers during these times. Therefore the implemented mechanism for updating patch qualities appears to follow the expected behavioural pattern.

When bees calculate action values they not only take into account their perception of the state of the environment in terms of the qualities of the patches and the time penalties required to move between them, but also their own state in terms of their remaining capacity to carry nectar. If a bee has only a small amount of nectar to collect to fill to capacity, then all but the most trivial of inter-patch movements would be considered non-optimal. Similarly, as the remaining nectar carrying capacity increases, the travel penalty required to fly to a candidate patch becomes an increasingly trivial component of the action value calculation. I would therefore expect that bees who were equipped with only small nectar carrying capacities would exhibit minimal levels of inter-patch movement compared to bees that were equipped with larger capacities, all else being equal.

We would also expect to see an increase in inter-patch traffic levels if the number of available patches within a landscape were increased. Inter-patch movements can only be instigated when the estimated quality of the current patch falls below the estimated quality of an alternative patch in the landscape, so an increased number of alternatives would typically increase levels of inter-patch traffic, as the probability of a preferable alternative being available would increase.

To test for these expected trends in traffic, I ran a series of trials in the model, in which I supplied bees with various fixed nectar carrying capacities and placed them in landscapes that varied in the number of patches available. Capacities of 10, 25, 125, 750 and 1,000 nectar units were trialled in landscapes containing 3, 4, 5, 7, 10 and 14 patches. 100 bees were allowed to forage simultaneously and mean results were taken across the colony and across the 10 trials of

each trial set. A separate trial set was carried out for each combination of capacity and landscape size tested.

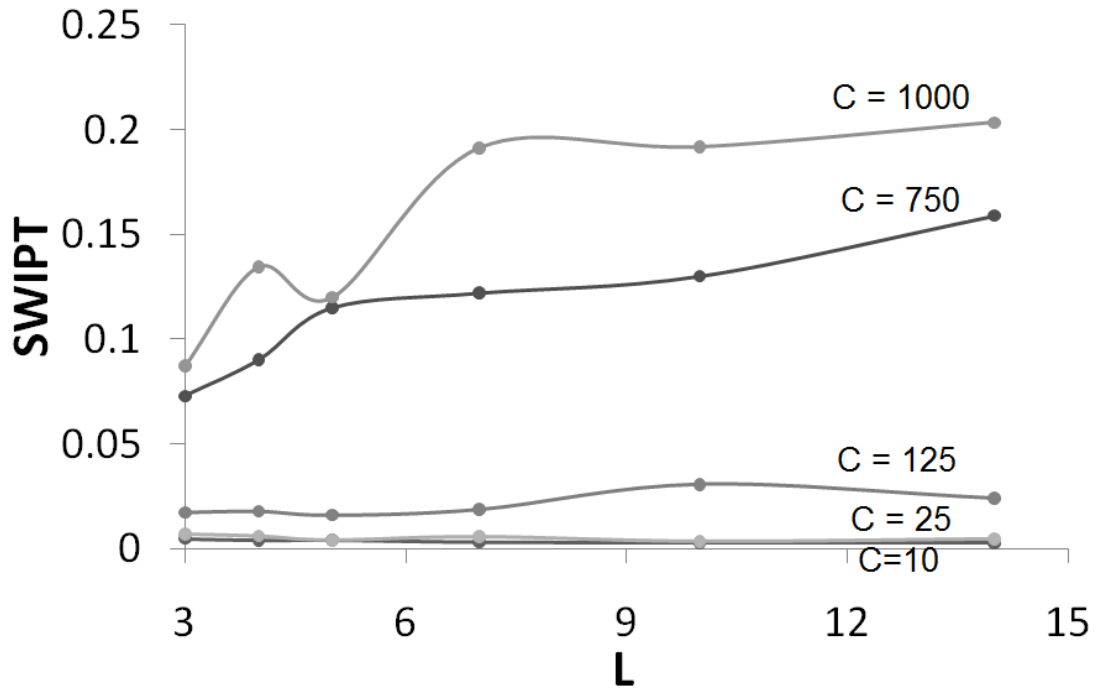

Figure 2.3. System-wide inter-patch traffic (SWIPT) with variable fixed nectar carrying capacity (C) and landscape size (L). X-axis shows landscape size in terms of the number of fields simulated, whilst Y-axis shows the average number of transitions per foraging bout made by bees in the colony. Separate plots for various nectar carrying capacities. 100 bees monitored over 10 replicate trials for each parameter combination.

I found for all landscape sizes tested, there was a clear increase in traffic levels as nectar carrying capacity increased (figure 2.3). I also found that by increasing the number of available patches in the landscape, traffic levels typically showed a general increase (figure 2.3). This increase is typically less pronounced when bees have smaller nectar carrying capacities, because the increased significance of inter-patch movement penalties in these scenarios means that movements away from the current patch are rarely considered beneficial, regardless of the number of alternatives available.

2.7 A contemporaneous application of the RL approach by others

Key ideas from my modelling approach that I have outlined in this chapter have also recently been used by Groβ *et al* (2008) to analyse optimality of a forager that must learn its

environment. I briefly discuss this adoption of my approach here, and draw attention to the differences that individuate each study.

In the model of Groß et al, as in HARVEST, the agent learns by maintaining estimates of the probability of reward for each action, and these estimates are updated using the Linear Operator learning rule. The simulation was configured to provide the agents with one of two decision rules, which are equivalent to the RL concept of a policy that I use in my model. The first decision rule is called MATCH and dictates that an agent chooses an action randomly, but the probability of selecting each action is proportional to the estimated reward expected to be obtained if the action is chosen. The second decision rule is called MAXIMISE and, like HARVEST, states that a forager should always choose the action that is estimated to offer the highest probability of reward.

The general similarities with my model are clear, but, unlike in HARVEST, Groß et al's model does not consider certain key influences, such as the internal state of the forager (i.e. whether it has begun to fill to capacity not), the state of the foraging environment (i.e. whether the current patch has been depleted), and the foraging landscape is described by resource probability distributions, rather than being spatially explicit. Thus, the model of Groß et al enables an abstract economic analysis of the performance of their foraging agent, but yields no information about spatial patterns of its movement. Therefore, despite certain similarities in formulation, their model is not applicable to our problem. Nevertheless, their model approaches economic optimality under fairly widespread conditions, as with HARVEST (see Chapter Three), which supports the likely effectiveness of the forager constructed under my similar approach.

2.8 Summary

Having argued for the validity of HARVEST as a model of forager movements, I will subsequently use it to calculate predictions of bumble bee-mediated gene flow levels. Using a mathematical model of pollinator-mediated gene flow that has been previously applied to bumble bee pollinators, my model can generate inter-patch traffic and patch residence predictions to populate the gene flow model. As well as providing an assessment of the influence of bumble bees as a pollen vector in configured landscapes and allowing gene flow predictions to be made at the empirically infeasible landscape scale, this approach also may mean that the behavioural processes that lead to these gene flow events can begin to be understood from an economic perspective. This allows us to explore alternative strategies for

configuring landscapes that, for example, minimise gene flow levels. We can also identify those scenarios that maximise gene escape risk in cases where gene flow levels need to be minimised.

In subsequent chapters I show how my model has been implemented to assess predictive capabilities for iconic behavioural traits related to bumble bee-mediated gene flow, more specific validation in terms of comparison with my own empirical investigations and extrapolations to the landscape-scale so that gene flow containment strategies can be assessed. First, however, I explore the economic performance of the foragers under a range of conditions (Chapter Three).

Chapter Three : Paper - A reinforcement learning solution to economically-motivated patch choice: application to landscape-scale bumble bee foraging

Statement of Contribution

The following paper was intended for submission to the journal Proceedings of the Royal Society B. I, as lead author, was primarily responsible for planning and writing the paper, and for designing and conducting the experiments detailed therein. My co-authors offered guidance and editing notes for revision of the manuscript, and provided constructive feedback about the experimental designs.

A reinforcement learning solution to economically-motivated patch choice: application to landscape-scale bumble bee foraging

Daniel Chalk[1*], Richard M. Everson[2] and James E. Cresswell[1]

[1]School of Biosciences, University of Exeter, Hatherly Laboratories, Prince of Wales Road, Exeter, EX4 4PS, United Kingdom; [2]School of Engineering, Computer Science and Mathematics, Harrison Building, University of Exeter, EX4 4QF, United Kingdom.

[*]Author for correspondence: d.chalk@ex.ac.uk

For submission to : Proceedings of the Royal Society B

ABSTRACT

We created a foraging model for an individual rate-maximiser capable of operating in spatially-explicit foraging scenarios involving patch choice. Our model combined a classical linear operator learning rule with a mechanism from reinforcement learning that quantifies the expected payoff of behavioural actions. We thereby formulated a patch choice strategy that considered the economic implications of information acquired through trial-and-error learning.

We applied our model to simulate bumble bees (*Bombus* spp.) foraging in a landscape that comprised fields of the mass-flowering crop *Brassica napus* (oilseed rape or canola). We also modelled the level of field-to-field gene flow mediated by the bees' movements. The simulated bees that used our reinforcement learning (RL) algorithm achieved near-optimal foraging performance as compared to omniscient foragers in a wide range of scenarios. The simulated RL bees had their lowest relative performance and moved most frequently between fields when the landscape contained many fields of similar low quality. Generally, however, our simulations predicted very low levels of bumble bee-mediated field-to-field gene flow in *B. napus*, because the frequency of the bees' field-to-field movements was very low compared to the frequency of flower-to-flower movements within the same field.

INTRODUCTION

Gene flow is the movement of genes from one population to another. It is important to understand gene flow, because it is one of the four factors that can change the allele frequency in a population, alongside mutation, genetic drift and natural selection, and it therefore plays a fundamental role in evolutionary processes, such as local adaptation and population differentiation. Additionally, concerns about the ecological impact of human activities make the study of gene flow important, because it can help in understanding the possible effects of anthropogenic habitat fragmentation on viability of the small or isolated populations that result, and in predicting the spread of transgenes from genetically modified (GM) crops.

In plants, genes are dispersed through both pollen and seeds, but cross-pollination is probably the principal means of gene dispersal for many plant species. New approaches to the analysis of extant patterns of genetic variation have enabled scientists to discern patterns of gene flow by cross-pollination and to begin to identify their governing influences. Nevertheless, it would be valuable to have theoretical models that can provide unifying generalizations and make predictions about cross-pollination and gene flow in new instances. Many plant species depend principally on animal pollinators for cross-pollination and, therefore, the pollinators' flower-to-flower movements dictate the spatial patterns of pollen dispersal and pollen-mediated gene flow. At present, theoretical approaches to pollinator movements are in their early stages. Previously, theoretical analyses (Ferrari *et al* 2006; Robledo-Arnuncio & Austerlitz 2006) and computer simulations (Campbell 1985; Strickler & Vinson 2000) have addressed pollinator-mediated gene flow only at small scales, and none of these were based in the foraging economics of patch choice or were applicable at small scales (i.e. for plants separated by metres rather than kilometres). Here, we begin to develop a new theoretical approach to modeling pollinator movements and estimating landscape-scale pollinator-mediated gene flow.

In principle, pollinator movements among patches of flowers can be modelled by existing economics-based foraging theory, which assumes that natural selection has promoted behaviour that maximises the profits of the foraging animal (Emlen 1966; MacArthur & Pianka 1966). The problems of which resource patch to exploit and how long to remain in each are well known in foraging theory as the 'patch choice' problem and the 'patch departure' problem respectively (Stephens & Krebs 1986). To solve these problems, theorists often assume that the forager holds complete information about patch quality. For example, the economically optimal behaviour for the 'patch departure' problem can be readily found using the Marginal Value

Theorem (Charnov 1976), but only if the forager has complete information. In contrast, solving either problem becomes complicated under the reasonable assumption that foragers do not have perfect knowledge of their environment (Pyke 1984). If the forager holds only incomplete information, it must not only exploit its environment, but also explore it in order to acquire the information necessary to make profitable decisions (Hirvonen *et al.* 1999; Keasar *et al.* 2002; Olsson & Brown 2006). Evaluating the most efficient compromise between exploration and exploitation complicates theoretical modelling. Alternative analytical approaches to patch choice are problematic in various ways. Some apply only to one dimensional habitats (Klaassen et al. 2006). Some necessitate the use of complex probability distributions (Mangel & Clark 1986; Rodriguez-Girones & Vasquez 1997) or techniques such as Fuzzy Logic (Inglis et al. 2001) that require many additional parameters that can be difficult to estimate realistically. Others are restricted to particular foraging scenarios by the condition that the forager cannot return to a patch once it has been initially rejected (Hoyle & Cresswell 2007b). In principle, the economically optimal behaviour can be found by using a computer and a dynamic programming procedure. However, the practicality of this rapidly decreases as the complexity of the landscape increases.

Computer simulation of the behaviour of individual foragers provides a feasible and flexible alternative to analytical approaches via the use of 'individual-based models' (Grimm & Railsback 2005). Previously, individual-based models for learning foragers facing the patch choice problem have either only partially relaxed the assumption of complete information (Bernstein *et al.* 1988), or focused on testing the equilibrium distributions of the foragers across the landscape (Bernstein *et al.* 1988; Ward *et al.* 2000; Hassall & Lane 2005; Fauvergue *et al.* 2006; Hancock & Milner-Gulland 2006), usually in order to investigate the Ideal Free Distribution (Fretwell & Lucas 1970). We are instead interested in the behavioural details of the inter-patch transitions that are made by individuals as they learn about an uncertain landscape. We therefore present an individual-based simulation model of an economically-motivated agent that faces a classic patch choice problem.

We have developed a spatially explicit patch choice model for incompletely informed central place foragers based on a decision rule that uses the principles of Reinforcement Learning (Sutton & Barto 1998). Reinforcement learning (RL) is a branch of artificial intelligence in which an agent makes decisions to maximise rewards. A reinforcement learning agent is incompletely informed about its foraging environment, because it remains unaware of the optimal solution to the problem it faces, but it can improve its performance by trial-and-error learning. This is an appropriate approach for the problem of animal foraging, because animals

typically exhibit adaptability when faced with uncertain environments (Dall *et al.* 2005). Here, we show that a forager with only simple cognitive capabilities and incomplete information can approach closely the performance of an omniscient forager under a range of realistic environmental scenarios.

We applied our model to simulate the foraging behaviour of bumble bees (*Bombus* spp.) among the patches of flowers where they collect nectar and pollen. The evolutionary fitness of a bumble bee colony depends on the rate at which nectar and pollen are accumulated in the nest (Heinrich 1979) and therefore we equate economic performance with the efficient harvesting of floral resources. The foraging behaviour of bumble bees has been widely studied (Goulson 2003) and there is evidence that individual bumble bees have a kilometre-scale foraging range (Cresswell *et al.* 2000; Westphal *et al.* 2006), but empirical investigations have failed to provide data on the details of their landscape-scale movements, because of technical limitations (Osborne *et al.* 1999). Although they are social, bumble bees forage by individual initiative (but see Chittka & Leadbeater 2005) and so it is impossible to deduce their foraging sites by decoding a 'dance language', as is done for honey bees, *Apis mellifera* L. (Visscher & Seeley 1982). As in other instances where landscape-scale movements have proven difficult to measure (Johst *et al.* 2001), theoretical modelling may therefore prove valuable. In this study, we present a model for the landscape-scale movements of incompletely-informed bumble bees (*Bombus* spp.) based on principles from Reinforcement Learning (RL). We consider only resource-collecting behaviours, because bumble bee workers generally are not distracted by other demands such as mating and predator avoidance (Heinrich 1983).

A further imperative for developing a theory of bumble bee movement is that bees are pollinators of the plants they visit. Pollination is an important mechanism of gene dispersal (Fenster 1991) and the level and spatial extent of cross-pollination may influence the genetic cohesion of plant species (Goodell *et al.* 1997), which can influence local adaptation and evolution (Slatkin 1985). The study of gene dispersal by insect cross-pollination also has practical importance because of concerns about the spread of engineered genes from genetically modified (GM) crops into conventional varieties (Poppy & Wilkinson 2005). We therefore used our model to simulate the foraging behaviour of bumble bees in fields of a plant with GM varieties, *Brassica napus* L. (oilseed rape or canola), where they collect nectar and pollen from the flowers (Cresswell 1999).

METHODS

The general model

The foragers in our model inhabit a landscape that contains patches of resource. Our foragers are completely informed about the locations of the patches in the landscape. Many foraging animals maintain a spatial representation of the landscape (e.g. Smith & Dawkins 1971) and we assume that this learning phase is already complete and that the forager has only to learn about the quality of the patches in the landscape. Each patch consists of an indefinitely large number of prey items, but the values of the items can differ. Consequently, patches can vary in quality because of variation in values of the prey items that they contain. For simplicity, we assume that each prey item has a net worth of either one resource unit (a good item) or zero units (a poor item), and we therefore define the quality of a patch by the proportion of good items that it contains. Once in a patch, the forager encounters prey items stochastically so that patch quality also defines the probability that a prey item encountered by the forager is good.

Each forager's sole objective is to become satiated with resources as quickly as possible and therefore it is a 'time minimizer' (Schoener 1971). Once the forager's capacity for resource intake is reached it returns home, ending the foraging bout. To evaluate foraging performance, we use the classic rate of gain equation (Stephens *et al*, 1986):

$$R = \frac{E_C}{T} \qquad \text{(Eq 1)}$$

where R is the net rate of resource gain, E_C is the amount of resource units required to satisfy the forager's capacity and T is the length of the foraging bout. As the forager's capacity, E_C, is fixed, the objective of our rate maximizing forager is to minimize the length of the foraging bout, T.

Foragers maintain estimates of the quality of each patch, and they choose the patch in which to forage by reviewing a list of inter-patch moves that can be made from the current location (including staying in the current patch). We define these moves as 'actions'. Each action also has an associated 'action value' that quantifies the expected economic consequences of choosing

that action. For simplicity, we neglect any differential between the cost of travel and harvesting, so that we can express the action value in units of time as the sum of the travel time and the expected time to fill to capacity at the destination. Thus, the action value associated with the action of moving from patch *i* to patch *j* is defined as

$$v(i, j) = \frac{c}{q(j)} h + t(i, j) \qquad \text{(Eq 2)}$$

where *c* is the forager's remaining capacity for resource intake, *q*(*j*) is the forager's estimate of the quality of patch *j* (i.e. the proportion of good prey items), *h* is the handling time per prey item and *t*(*i*, *j*) is the travel time between patch *i* and patch *j*. The forager's policy is always to select the action with the lowest associated action value, because this is the forager's current best guess as to the course of action that will lead most rapidly to satiation. A selection is made after every prey item that the forager eats. Foragers are free to revisit patches.

The forager learns by using its experience of sampling prey items to update its 'personal information' (Dall *et al.* 2005) about action values. Good prey items improve the forager's perception of the patch and poor prey items cause it to worsen. In this model we adopt the widely used 'linear operator' learning rule (Bush & Mosteller 1951; Kacelnik & Krebs 1985; McNamara & Houston 1987), which has performed well in a variety of foraging situations (Beauchamp 2000). Formally, for a given forager

$$q(i)_{new} = \beta s(i) + (1 - \beta) q(i)_{current} \qquad \text{(Eq 3)}$$

where $q(i)_{new}$ is the updated estimated quality of patch *i*, $q(i)_{current}$ is the current estimated quality of patch *i*, *s(i)* is the resource value of the newly sampled prey item sample in patch *i*, and β is a sensitivity parameter (with $0 \leq \beta \leq 1$). Thus, every time the forager samples a prey item in patch *i*, it updates the list of action values and evaluates in which patch it should forage. When foragers eat many prey items successively, the value of β effectively determines the span of the forager's memory for prey values, because the forager's sensitivity to newly encountered prey items is determined by β. When β = 1, the forager ignores previous experiences in the patch and

when $\beta = 0$, the forager ignores new experiences. For all intermediate values, increasing β increases the importance of the latest prey item sample.

For comparison, we calculated the foraging performance of otherwise equivalent omniscient foragers, whose list of action values is calculated (Eq 2) using the actual patch qualities.

Application to bumble bees and **Brassica napus**

We ran experimental simulations with our bees facing small, medium and large landscapes that contained either three, seven or 14 fields of *B. napus*, which represent realistic numbers of foraging locations faced by bumble bees (Osborne *et al.* 1999). A field of *B. napus* blooms for several weeks (Hoyle & Cresswell 2007a) and so it is reasonable to assume that bees have already learned the spatial locations of these persistent resources. All inter-field distances and distances to and from the nest were assumed to be 1 km, and bees were assumed to fly at 7 m s^{-1} (Osborne *et al.* 1999), which implies travel times of approximately 143 s for each of these journeys. The resource capacity of the bees has an impact on their behaviour and we therefore investigated small, medium and large capacities as follows. Bumble bees have been observed to visit approximately 500 flowers in an agricultural field of *B. napus* (Cresswell *et al.* 2002). We therefore assumed that a good flower contained a reward of one unit of nectar and/or pollen and we investigated bees with capacities of either 50, 500 or 1,000 units of floral resource (nectar and pollen). We varied bee capacity as a proxy for landscape richness, because the requirement to harvest more good items to reach capacity is equivalent to foraging in a landscape where good food items are individually less rewarding. Once in a field, we assumed that a bee required 3 s to visit a flower and extract its resources, if any (Cresswell 1999). Bees foraged for 8 h.

Within a field, each flower contained either one or zero units of floral resource. In reality, the amount of resources in a single flower is drawn from a continuous distribution, but we believe that this simplification is tolerable, because previous studies have shown that bumble bees behave as if they categorize floral rewards on a binary scale, i.e. as either above or below a given threshold (Hodges 1985b). Additionally, we do not model depletion or replenishment of floral resources in this study for three reasons. First, in the UK, commercial fields of *B. napus* flower in April and May when bees are scarce, and only about 10% of the millions of flowers in each field receive a bee visit in their blooming period (Hayter & Cresswell 2006), which implies

that depletion by bees is almost incrementally small. Second, nectar production in *B. napus* decelerates with nectar level to produce a consistent level of nectar reward in undepleted flowers (J.E. Cresswell, personal observation). Third, when we simulated higher abundances of bees and depletion effects became important, the resource levels of the patches invariably became homogenised, because bees initially concentrated on the most rewarding fields and the patch choice problem evaporated. However, if desired, our investigations of static patch qualities can be seen as snapshots of the system's transient behaviour as it makes its way to the homogeneous-quality equilibrium.

In each landscape, the quality of a patch of flowers, or field, is determined by the proportion of flowers that contain a good reward, G. The lowest and highest quality fields were assigned qualities symmetrically about a mean of $\overline{G}$ = 0.5 and we characterized the heterogeneity of a landscape by the 'resource range', G^*, which was the difference in the quality between the best and worst quality fields, i.e. $G^* = G_{best} - G_{worst}$. Once we had set the resource range for a landscape, all remaining fields were assigned qualities randomly from a uniform distribution taken over the resource range. We investigated the effect of changes in landscape resource variability on bee behaviour by testing resource ranges varying from $G^* = 0.0$ (the homogenous landscape) to $G^* = 0.8$ (the most variable landscape).

Whilst bumble bees are individual foragers, their colony is social and they combine their harvests into a collective store. The foraging behaviour of an individual worker bee is influenced by colony resource levels (Cartar 1992; Dornhaus & Chittka 2005). We therefore provided our model bees with initial action values where the estimates of field quality were based on the average quality of the landscape, which is $\overline{G}$ = 0.5.

We set the learning rate parameter (see Eq 2) as $\beta = 0.05$ following a β-optimisation study (Appendix A), because this value yielded very good foraging performance across a wide variety of scenarios, which is consistent with other studies where low values of the learning rate parameter performed well (Keasar *et al.* 2002).

We model the level of pollinator-mediated gene flow into a field, ξ, as

$$\xi = \frac{E\psi}{b} \qquad \text{(Eq 4)}$$

where E is the proportion of bees that arrive in the field carrying pollen from another field, ψ is the number of flowers that each bee is capable of fully fertilizing with pollen from another field and b is the mean number of flowers that bees visit in the field (Cresswell *et al.* 2002). Values for both E and b emerged from our simulations and we set $\psi = 1$ based on published empirical results (Cresswell *et al.* 2002). To evaluate the potential risk of genetic escape from a single GM field via cross-pollination, we attributed the source of the GM gene flow to the field with the highest number of foragers emigrating to another field per day. We then described the resulting gene flow by the mean value of ξ across the non-source fields, $\bar{\xi}$. Across all the simulations, the highest possible rate of gene flow is conservatively approximated by the maximum value of

$$\xi_{\max} = E_{\max}(n_{\max} - 1)\overline{\xi_{\max}} \qquad \text{(Eq 5)}$$

where $E_{\max}$ is the highest observed value of E, $n_{\max}$ denotes the largest number of fields in the landscape, and $\bar{\xi}_{\max}$ denotes the highest observed value of $\bar{\xi}$.

The simulation model was implemented using the Python programming language (version 2.4.2) and was executed using the Python interpreter. For each parameter value combination, we simulated 100 RL bees and 100 omniscient bees in each of 10 randomly generated landscapes.

RESULTS

Bumble Bee Behaviour

Bumble bees explored virtually all of the available fields only when the quality of fields was fairly even, and the proportion of fields explored decreased with increasing landscape size (Fig 1). Bumble bees explored their maximum number of fields relatively quickly, often after only around 10% of the duration of the 8 h foraging day (Fig. 1). Indeed, in landscapes with the most variation in field quality, bees typically exploited the same field continuously after the first foraging bout. By contrast, in landscapes with relatively even field qualities, bumble bees often visited several fields during each bout. Thus, the frequency of inter-field travel decreased with increasing variability in field quality (Fig. 2). Bees in poor quality landscapes moved between fields more frequently than those in high quality landscapes (Fig. 3). Generally, the frequency of inter-field travel increased with landscape size.

Foraging Performance

As expected, the economic performance of bees (i.e rates of resource gain) increased during the initial phase of foraging, because the bees learned the location of better-than-average fields. Performance improved most quickly in landscapes with highest variation in field qualities, G^*. The bees' absolute levels of performance also increased with variability in field qualities, because of the greater value of the highest quality fields.

The performance of bees approached that of omniscient foragers fairly closely under a wide range of scenarios (Fig. 4), but bees were comparatively weak in two particular kinds of scenario. First, bees performed relatively poorly in resource-poor landscapes where the fields were similar in quality (bottom right corner of Fig. 4). Second, bees performed slightly less well in rich landscapes when fields were highly variable in quality (top left corner of Fig. 1).

Predictions of pollinator-mediated gene flow

Our simulations predict very low levels of bumble bee-mediated field-to-field gene flow in *B. napus* (Fig 5a) and ξ_{max} = 0.05% (Eq 5). Levels of gene flow are low for two reasons. First, the probability that a bee arrives bearing pollen from the source field, *E*, is much lower than 1/*n* (Fig 5b), where *n* denotes the number of fields in the landscape, because the bees most often arrived at a field directly from their nest. This was particularly true in rich landscapes and/or landscapes with high variations in field quality. Second, bees typically visited many flowers during a bout in a single field (Fig 5b), which meant that the great majority of the pollinating flower-flower movements made by bees were within the same field. Consequently, the value of *b* (Eq 4) was high and within-field pollinations predominated, because any incoming pollen from another field is rapidly exhausted over only a few successive pollinations (Cresswell *et al*, 2002). However, the influences of variation in quality among fields, landscape size and overall landscape richness interact, and it is difficult to make simple generalizations about the influences of single factors (Fig 5a).

DISCUSSION

Using a reinforcement learning (RL) algorithm, our simulated bees exhibited two behaviours that are characteristic of real bees and other floral foragers that forage by individual initiative rather than being socially directed like honey bees. First, the RL bees focused their foraging on the more rewarding among patches of variable quality (Cartar 2004; Burns & Thomson 2006). Second, RL bees returned directly to the same high value patch on beginning successive foraging bouts (Isnec *et al.* 1997), which enabled them to exhibit site fidelity (Osborne & Williams 2001). In addition, the site fidelity of RL bees was more pronounced at higher levels of variation in patch quality, which has been observed in birds (Dow & Lea 1987). This latter phenomenon remains to be demonstrated empirically in floral foragers, although it is reasonable to anticipate it, because an increased differential in quality among patches is likely to aid an economically motivated animal in making a behavioural differentiation among them. Based on these comparisons, we tentatively suggest that the RL foraging model begins to provide a basis for the emulation of bee foraging behaviour, although we remain somewhat cautious for reasons outlined below.

The foraging behaviour of worker bumble bees is undoubtedly directed towards promoting a large and rapid input of harvested floral resources into the parental colony (Heinrich 1979). Unsurprisingly, therefore, economic models based on the maximization of foraging rate have successfully predicted many aspects of bee foraging behaviour at small spatial scales (Best & Bierzychudek 1982; Hodges 1985a; Dreisig 1995). Therefore, the RL model is credibly applied to simulate bees at the landscape scale in the limited sense that it offers a high rate of economic return. In our simulations, the RL bees achieved near-optimal solutions to patch choice problems under a wide range of landscape scenarios. Initially equipped with only an estimate of the average landscape quality, and with no other prior information about variation in field quality, the RL bees used trial-and-error reinforcement learning to decide which field to exploit. Despite these cognitive constraints, RL bees often achieved over 90% of the rate of resource collection of otherwise equivalent omniscient foragers. In effect, the RL bees usually had little difficulty in responding to stationary, systematic differences in field quality despite the statistical error inherent in a process of estimation that was based on receiving small pieces of binary information (good vs. poor flowers), in part because of the low weight given to the most recently sampled flower. Compared to omniscient foragers, the RL bees were weakest when the systematic differences in field quality were small and therefore similar in magnitude to the sampling error in the estimates of field quality, because RL bees could falsely perceive their current field as inferior based on an unlucky run of bad luck and thereby incur unnecessary travel costs. This kind of mistake was seemingly most costly in poor landscapes, because of greater comparability in the costs of exploration and exploitation. However, in poor landscapes, even the omniscient forager had a low rate of reward uptake because only average quality fields were available and so the proportionately lower performance of the RL bee was a small penalty in absolute terms. RL bees also showed some relative weakness in larger landscapes, because they sometimes failed to explore the landscape exhaustively and find the best possible patch, although they normally identified a relatively high quality patch. Overall, our results have tested and supported the applicability of the RL model to bees insofar as its failure to make adequate economic returns would have discredited it.

We do not yet know whether our simple RL model will emulate real bumble bee behaviour at the landscape scale, because there are as yet no data for comparison. However, our model is based on some plausible postulates. As in our model, real bees are very quick learners (Dukas & Real 1993) and capable of spatial memory (Burns & Thomson 2006). It is reasonable for us initially to assume a purely economic foraging motivation, because the influence of factors such as risk of predation is considered to be fairly minimal (Pyke 1979), although such considerations could be included as extensions to the core model. The naive bee's initial assessment of average landscape quality fits the observation that bees adapt their behaviour

according to the resource status of their colony (Cartar 1992; Dornhaus & Chittka 2005). Furthermore, we have avoided imposing certain *ad hoc* mechanisms that are sometimes incorporated into foraging models, such as enforced periodic sampling (Thuijsman *et al.* 1995). Indeed, a mechanism for periodic sampling emerged serendipitously in our simulation, because RL bees occasionally left a relatively high value patch to explore elsewhere, because they had encountered a sequence of poor rewards that had originated stochastically as a run of bad luck.

Based on the simulations of the foraging movements of RL bees, we predict that the level of bumble bee-mediated field-to-field gene flow will be very low. Low levels of gene flow emerged because of the strong site fidelity of RL bees and its associated long residence times in fields, which favour the predominance of within-field pollination. Collectively, our simulations predict an upper bound in the order of 0.05% of a field's seed being fertilized by bee-mediated gene flow from a designated GM source field. However, this boundary was approached only when the differential in quality among fields was small. Thus, the boundary could be approached in systems where bees depleted and thereby homogenized field qualities, because this increases traffic levels as bees constantly switch from over-harvested fields. In most scenarios where variation in field qualities was maintained, gene flow was much lower than this. In reality, differentials in nectar levels have been found in agricultural fields of *B. napus* (K. E. Hayter and J. E. Cresswell, unpublished results) and bees are often too scarce to deplete the flowers more than incrementally in fields that bloom early in the year (Hayter & Cresswell 2006) while bumble bee colonies are small. In these cases, we predict only minute levels of bumble bee-mediated gene flow among fields of agricultural *B. napus*. Our results suggest that maintaining high variability among fields in the availability of floral resources, such as nectar and pollen, could form part of a strategy for the genetic containment of GM *B. napus*.

In this study, we have focused on the potential application of the RL foraging model to landscape-scale bumble bee behaviour. However, we envisage that the RL approach could also be useful for simulating the foraging behaviour of other animals at both small and large spatial scales, because many animals are energy maximisers (Emlen 1966) that learn the resource availabilities in their environment (Valone 2006) and possess a spatial memory of patch qualities (Milinski 1994; Hirvonen *et al.* 1999). Potentially, previously intractable patch choice problems could be approached by using the RL model, such as migratory behaviour or population dispersal. In summary, we believe that reinforcement learning principles may offer some new avenues for the simulation of economically motivated behaviours in animals.

ACKNOWLEDGEMENTS

Our thanks go to the BBSRC for funding this research.

REFERENCES

Beauchamp, G. (2000) Learning rules for social foragers: Implications for the producer-scrounger game and ideal free distribution theory. *Journal of Theoretical Biology*, **207**, 21-35.

Bernstein, C., Kacelnik, A., & Krebs, J.R. (1988) Individual decisions and the distribution of predators in a patchy environment. *Journal Of Animal Ecology*, **57**, 1007-1026.

Best, L.S. & Bierzychudek, P. (1982) Pollinator foraging on foxglove (*Digitalis purpurea*): a test of a new model. *Evolution*, **36**, 70-79.

Burns, J.G. & Thomson, J.D. (2006) A test of spatial memory and movement patterns of bumblebees at multiple spatial and temporal scales. *Behavioral Ecology*, **17**, 48-55.

Bush, R.R. & Mosteller, F. (1951) A mathematical model for simple learning. *Psychological Review*, **58**, 313-323.

Campbell, D.R. (1985) Pollen and gene dispersal: the influences of competition for pollination. *Evolution*, **39**, 418-431.

Cartar, R.V. (1992) Adjustment of foraging effort and task switching in energy-manipulated wild bumblebee colonies. *Animal Behaviour*, **44**, 75-88.

Cartar, R.V. (2004) Resource tracking by bumble bees: Responses to plant-level differences in quality. *Ecology*, **85**, 2764-2771.

Charnov, E.L. (1976) Optimal foraging: the marginal value theorem. *Theoretical Population Biology*, **9**, 129-136.

Chittka, L. & Leadbeater, E. (2005) Social learning: Public information in insects. *Current Biology*, **15**, R869-R871.

Cresswell, J.E. (1999) The influence of nectar and pollen availability on pollen transfer by individual flowers of oil-seed rape (*Brassica napus*) when pollinated by bumblebees (*Bombus lapidarius*). *Journal of Ecology*, **87**, 670-677.

Cresswell, J.E., Osborne, J.L., & Bell, S.A. (2002) A model of pollinator-mediated gene flow between plant populations with numerical solutions for bumblebees pollinating oilseed rape. *Oikos*, **98**, 375-384.

Cresswell, J.E., Osborne, J.L., & Goulson, D. (2000) An economic model of the limits to foraging range in central place foragers with numerical solutions for bumblebees. *Ecological Entomology*, **25**, 249-255.

Dall, S.R.X., Giraldeau, L.A., Olsson, O., McNamara, J.M., & Stephens, D.W. (2005) Information and its use by animals in evolutionary ecology. *Trends In Ecology & Evolution*, **20**, 187-193.

Dornhaus, A. & Chittka, L. (2005) Bumble bees (Bombus terrestris) store both food and information in honeypots. *Behavioral Ecology*, **16**, 661-666.

Dow, S.M. & Lea, S.E.G. (1987) Sampling of schedule parameters by pigeons: tests of optimizing theory. *Animal Behaviour*, **35**, 102-114.

Dreisig, H. (1995) Ideal free distibution of nectar foraging bumblebees. *Oikos*, **72**, 161-172.

Dukas, R. & Real, L.A. (1993) Effects of nectar variance on learning by bumble bees. *Animal Behaviour*, **45**, 37-41.

Emlen, J.M. (1966) The role of time and energy in food preference. *American Naturalist*, **100**, 611-617.

Fauvergue, X., Boll, R., Rochat, J., Wajnberg, E., Bernstein, C., & Lapchin, L. (2006) Habitat assessment by parasitoids: consequences for population distribution. *Behavioral Ecology*, **17**, 522-531.

Fenster, C.B. (1991) Gene flow in *Chamaecrista fasciculata* (Leguminosae) I. Gene dispersal. *Evolution*, **45**, 398-409.

Ferrari, M.J., Bjornstad, O.N., Partain, J.L., & Antonovics, J. (2006) A gravity model for the spread of a pollinator-borne plant pathogen. *American Naturalist*, **168**, 294-303.

Fretwell, S.D. & Lucas, H.L. (1970) On territorial behaviour and other factors influencing habitat distribution in birds. *Acta Biotheoretica*, **19**, 16-36.

Goodell, K., Elam, D.R., Nason, J.D., & Ellstrand, N.C. (1997) Gene flow among small population of a self-incompatible plant: An interaction between demography and genetics. *American Journal of Botany*, **84**, 1362-1371.

Goulson, D. (2003) *Bumblebees* Oxford University Press, Oxford, UK.

Grimm, V. & Railsback, S.F. (2005) *Individual-based modeling and ecology* Princeton University Press, Princeton, New Jersey, USA.

Hancock, P.A. & Milner-Gulland, E.J. (2006) Optimal movement strategies for social foragers in unpredictable environments. *Ecology*, **87**, 2094-2102.

Hassall, M. & Lane, S.J. (2005) Partial feeding preferences and the profitability of winter-feeding sites for brent geese. *Basic And Applied Ecology*, **6**, 559-570.

Hayter, K. & Cresswell, J. (2006) The influence of pollinator abundance on the dynamics and efficiency of pollination in arable *Brassica napus*: implications for landscape-scale gene dispersal. *Journal of Applied Ecology*, **43**, 1196-1202.

Heinrich, B. (1983) Do bumblebees forage optimally, and does it matter? *American Zoologist*, **23**, 273-281.

Hirvonen, H., Ranta, E., Rita, H., & Peuhkuri, N. (1999) Significance of memory properties in prey choice decisions. *Ecological Modelling*, **115**, 177-189.

Hodges, C.M. (1985a) Bumble bee foraging: energetic consequences of using a threshold departure rule. *Ecology*, **66**, 188-197.

Hodges, C.M. (1985b) Bumble bee foraging: the threshold departure rule. *Ecology*, **66**, 179-187.

Hoyle, M. & Cresswell, J.E. (2007a) The effect of wind direction on cross-pollination in wind-pollinated GM crops. *Ecological Applications*, *in press*.

Hoyle, M. & Cresswell, J.E. (2007b) A search theory model of patch-to-patch forager movement with application to pollinator-mediated gene flow. *Journal of Theoretical Biology*, ***in press***.

Inglis, I.R., Langton, S., Forkman, B., & Lazarus, J. (2001) An information primacy model of exploratory and foraging behaviour. *Animal Behaviour*, **62**, 543-557.

Isnec, M.R., Couvillon, P.A., & Bitterman, M.E. (1997) Short-term spatial memory in honeybees. *Animal Learning & Behavior*, **25**, 165-170.

Iwasa, Y., Higashi, M., & Yamamura, N. (1981) Prey distribution as a factor determining the choice of optimal foraging strategy. *American Naturalist*, **117**, 710-723.

Johst, K., Brandl, R., & Pfeifer, R. (2001) Foraging in a patchy and dynamic landscape: Human land use and the White Stork. *Ecological Applications*, **11**, 60-69.

Kacelnik, A. & Krebs, J.R. (1985). Learning to exploit patchily distributed food. In *Behavioural ecology (The 25th symposium of the British Ecological Society)* (eds R.M. Sibly & R.H. Smith), pp. 189-205. Blackwell Scientific Publications, Oxford.

Keasar, T., Rashkovich, E., Cohen, D., & Shmida, A. (2002) Bees in two-armed bandit situations: foraging choices and possible decision mechanisms. *Behavioral Ecology*, **13**, 757-765.

Klaassen, R.H.G., Nolet, B.A., Van Gils, J.A., & Bauer, S. (2006) Optimal movement between patches under incomplete information about the spatial distribution of food items. *Theoretical Population Biology*, **70**, 452-463.

Krebs, J.R., Kacelnik, A., & Taylor, P. (1978) Test of optimal sampling by foraging great tits. *Nature*, **275**, 27-31.

MacArthur, R.H. & Pianka, E.R. (1966) On optimal use of a patchy environment. *American Naturalist*, **100**, 603-609.

Mangel, M. & Clark, C.W. (1986) Towards A Unified Foraging Theory. *Ecology*, **67**, 1127-1138.

McNamara, J.M. & Houston, A.I. (1987) Memory and the efficient use of information. *Journal of Theoretical Biology*, **125**, 385-395.

Milinski, M. (1994) Long-term memory for food patches and implications for ideal free distributions in sticklebacks. *Ecology*, **75**, 1150-1156.

Olsson, O. & Brown, J.S. (2006) The foraging benefits of information and the penalty of ignorance. *Oikos*, **112**, 260-273.

Osborne, J.L., Clark, S.J., Morris, R.J., Williams, I.H., Riley, J.R., Smith, A.D., Reynolds, D.R., & Edwards, A.S. (1999) A landscape-scale study of bumble bee foraging range and constancy, using harmonic radar. *Journal of Applied Ecology*, **36**, 519-533.

Osborne, J.L. & Williams, I.H. (2001) Site constancy of bumblebees in an experimentally patchy habitat. *Agriculture, Ecosystems and Environment*, **83**, 129-141.

Poppy, G. & Wilkinson, M., eds. (2005) *Gene flow from GM plants*, pp 241. Blackwell Publishing, Oxford.

Pyke, G.H. (1979) Optimal foraging in bumblebees: rules of movement between flowers within inflorescences. *Animal Behaviour*, **27**, 1167-1181.

Pyke, G.H. (1984) Optimal foraging theory: a critical review. *Annual Review of Ecology and Systematics*, **15**, 523-575.

Railsback, S.F., Lamberson, R.H., Harvey, B.C., & Duffy, W.E. (1999) Movement rules for individual-based models of stream fish. *Ecological Modelling*, **123**, 73-89.

Ricketts, T.H. (2001) The matrix matters: Effective isolation in fragmented landscapes. *American Naturalist*, **158**, 87-99.

Robledo-Arnuncio, J.J. & Austerlitz, F. (2006) Pollen dispersal in spatially aggregated populations. *American Naturalist*, **168**, 500-511.

Rodriguez-Girones, M.A. & Vasquez, R.A. (1997) Density-dependent patch exploitation and acquisition of environmental information. *Theoretical Population Biology*, **52**, 32-42.

Schoener, T.W. (1971) Theory of feeding strategies. *Annual Review of Ecology and Systematics*, **11**, 369-404.

Slatkin, M. (1985) Gene flow in natural populations. *Annual Review of Ecology and Systematics*, **16**, 393-430.

Smith, J.N.M. & Dawkins, R. (1971) Hunting Behavior Of Individual Great Tits In Relation To Spatial Variations In Their Food Density. *Animal Behaviour*, **19**, 695-&.

Stephens, D.W. & Krebs, J.R. (1986) *Foraging Theory* Princeton University Press, Princeton, NJ.

Strickler, E. & Vinson, J.W. (2000) Simulation of the effect of pollinator movement on alfalfa seed set. *Environmental Entomology*, **29**, 907-918.

Sutton, R.S. & Barto, A.G. (1998) *Reinforcement learning* The MIT Press, Cambridge, Massachusetts, USA.

Thuijsman, F., Peleg, B., Amitai, M., & Shmida, A. (1995) Automata, Matching And Foraging Behavior Of Bees. *Journal of Theoretical Biology*, **175**, 305-316.

Valone, T.J. (2006) Are animals capable of Bayesian updating? An empirical review. *Oikos*, **112**, 252-259.

Visscher, P.K. & Seeley, T.D. (1982) Foraging strategy of honeybee colonies in a temperate deciduous forest. *Ecology*, **63**, 1790-1801.

Ward, J.F., Austin, R.M., & MacDonald, D.W. (2000) A simulation model of foraging behaviour and the effect of predation risk. *Journal Of Animal Ecology*, **69**, 16-30.

Westphal, C., Steffan-Dewenter, I., & Tscharntke, T. (2006) Bumblebees experience landscapes at different spatial scales: possible implications for coexistence. *Oecologia*, **149**, 289-300.

Figure legends

Figure 1. The cumulative number of unique fields visited by a bee (*y*-axis) *vs*. the number of time steps elapsed (*x*-axis) during an 8 h foraging trial. Dashed lines indicate scenarios with the most variation in field quality and continuous lines indicate totally even field qualities. Thick lines indicate curves relating to landscapes with 14 fields and thin lines indicate landscapes with three fields. Points are interpolated for ease of inspection only.

Figure 2. Mean number of field-to-field transitions per bee per foraging bout (y-axis) vs. variation in quality among fields, G*, (x-axis) for various sizes of landscape. Error bars indicate mean standard errors of individual bees from the set of bees in a trial.

Figure 3. Mean number of field-to-field transitions per bee per bout (y-axis) vs. landscape richness (x-axis). Higher capacities represent poorer landscapes. Error bars indicate mean standard errors of individual bees from the set of bees in a trial.

Figure 4. Contour plot showing the impact on foraging performance levels of resource variability, G*, (expressed as the range between the highest and lowest quality fields) and bee capacity (or landscape richness). Performance is expressed as the proportion of performance attained by equivalent omniscient foragers. Performance data averaged over 40 trials for improved precision.

Figure 5. (a) Mean proportion of flowers (or seed set) of a gene flow-sink field completely fertilized by incoming pollen (y-axis) vs. variation in quality among fields, G* (x-axis). (b) Mean probability of arriving in the sink field from a source field, E (y-axis) vs. variation in quality among fields, G* (x-axis). (c) Mean number of flowers pollinated by bees during a visit to the sink field, b, (y-axis) vs. variation in quality among fields, G* (x-axis). In all three panels, data are presented for rich and poor landscapes and small and large landscape sizes. Dashed lines indicate the richest landscapes and continuous lines indicate the poorest. Thick lines indicate curves relating to landscapes with 14 fields and thin lines indicate landscapes with three fields.

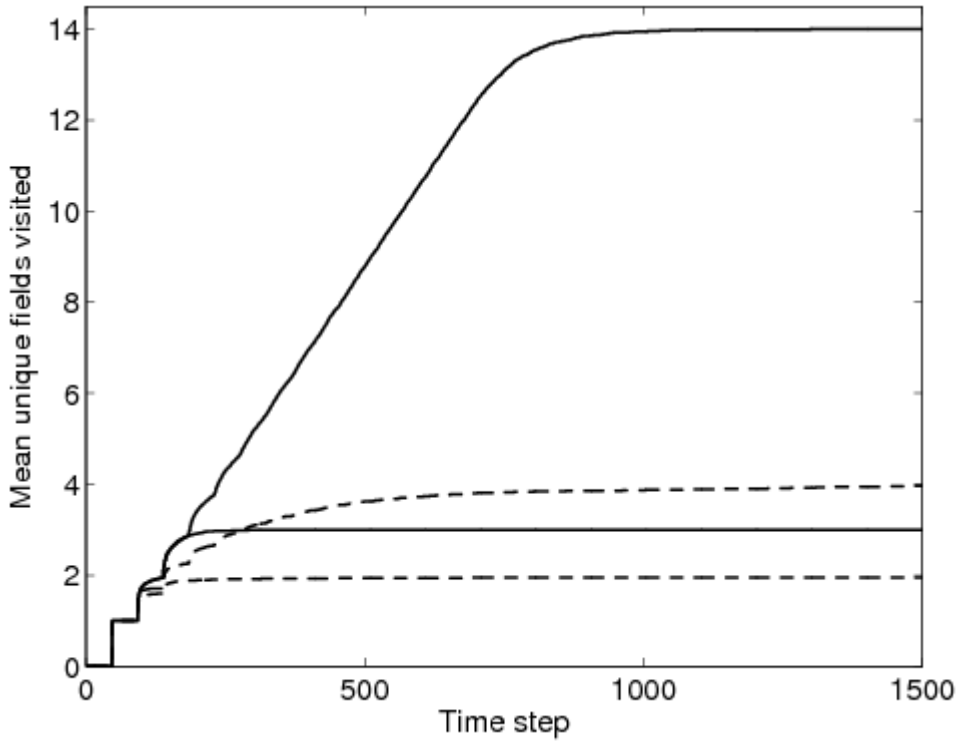
Mean unique fields visited
Time step
0
2
4
6
8
10
12
14
0
500
1000
1500

Figure 1

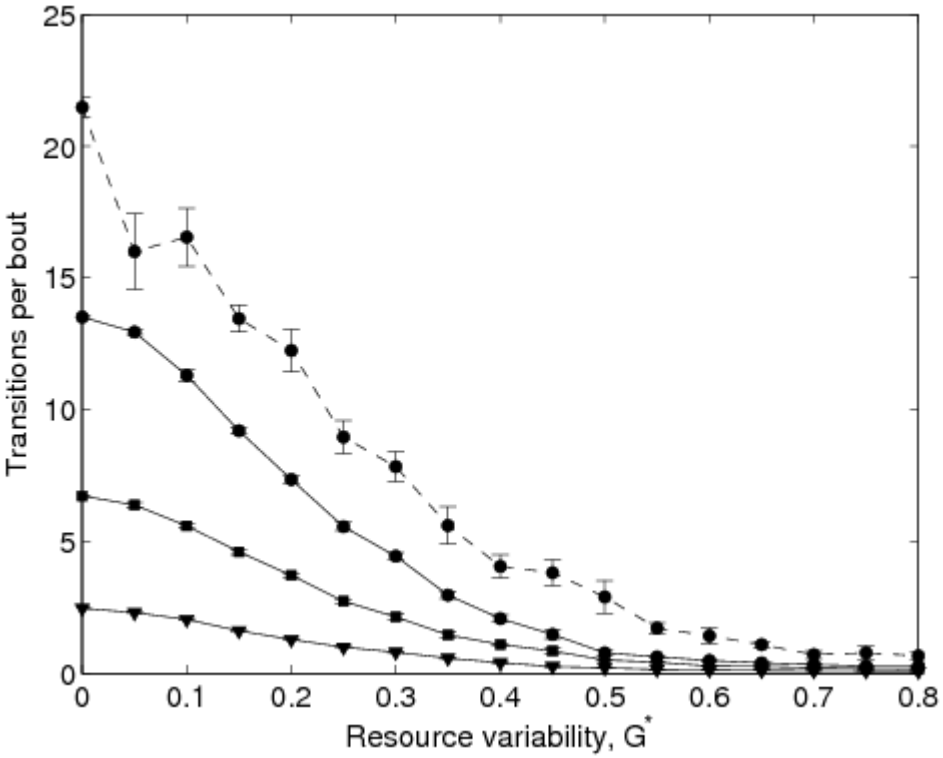
25
20
15
10
5
0
Transitions per bout
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Resource variability, G*

Figure 2

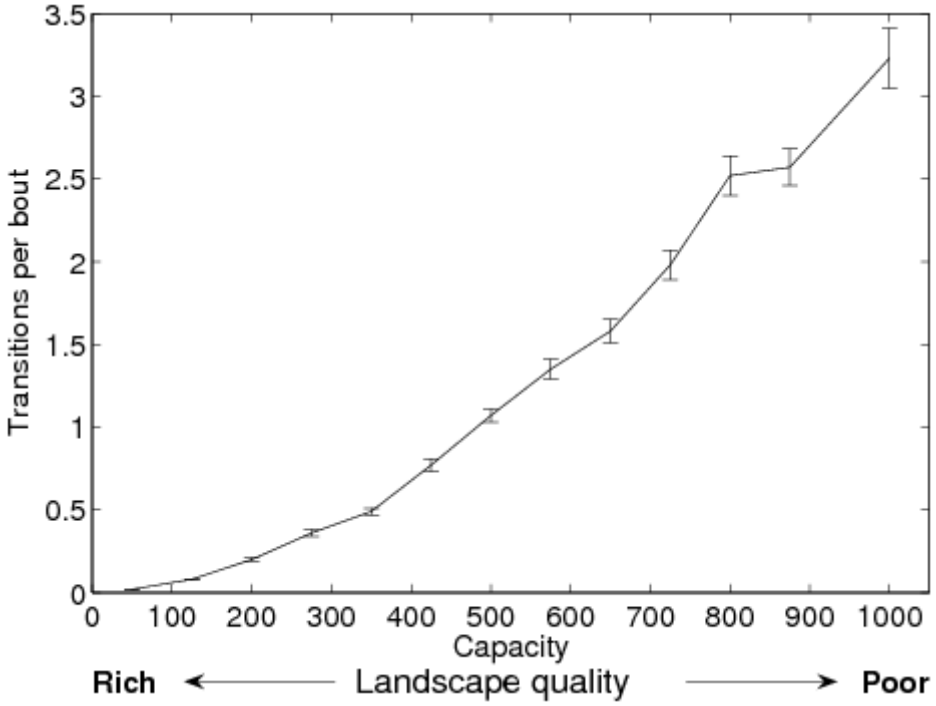
Transitions per bout
3.5
3
2.5
2
1.5
1
0.5
0
0
100
200
300
400
500
600
700
800
900
1000
Capacity
Rich
Landscape quality
Poor

Figure 3

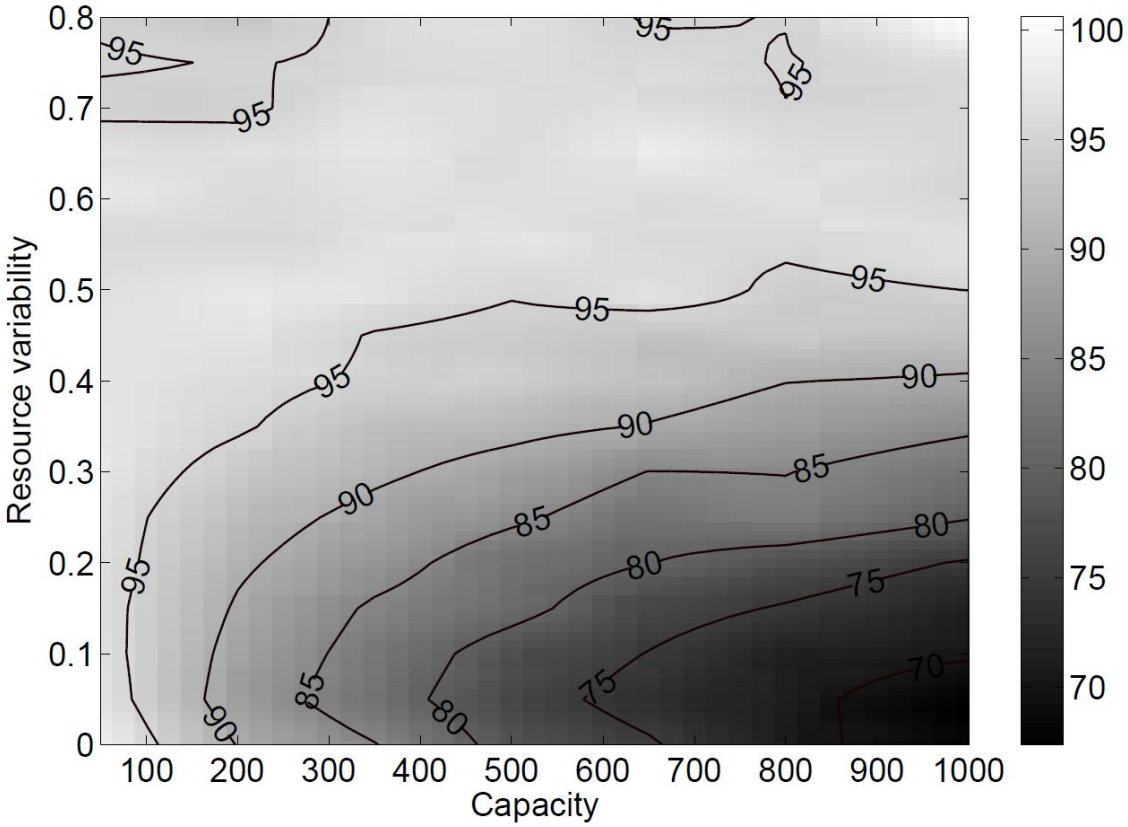
Resource variability
Capacity
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100
200
300
400
500
600
700
800
900
1000
95
90
85
80
75
70
100
95
90
85
80
75
70

Figure 4

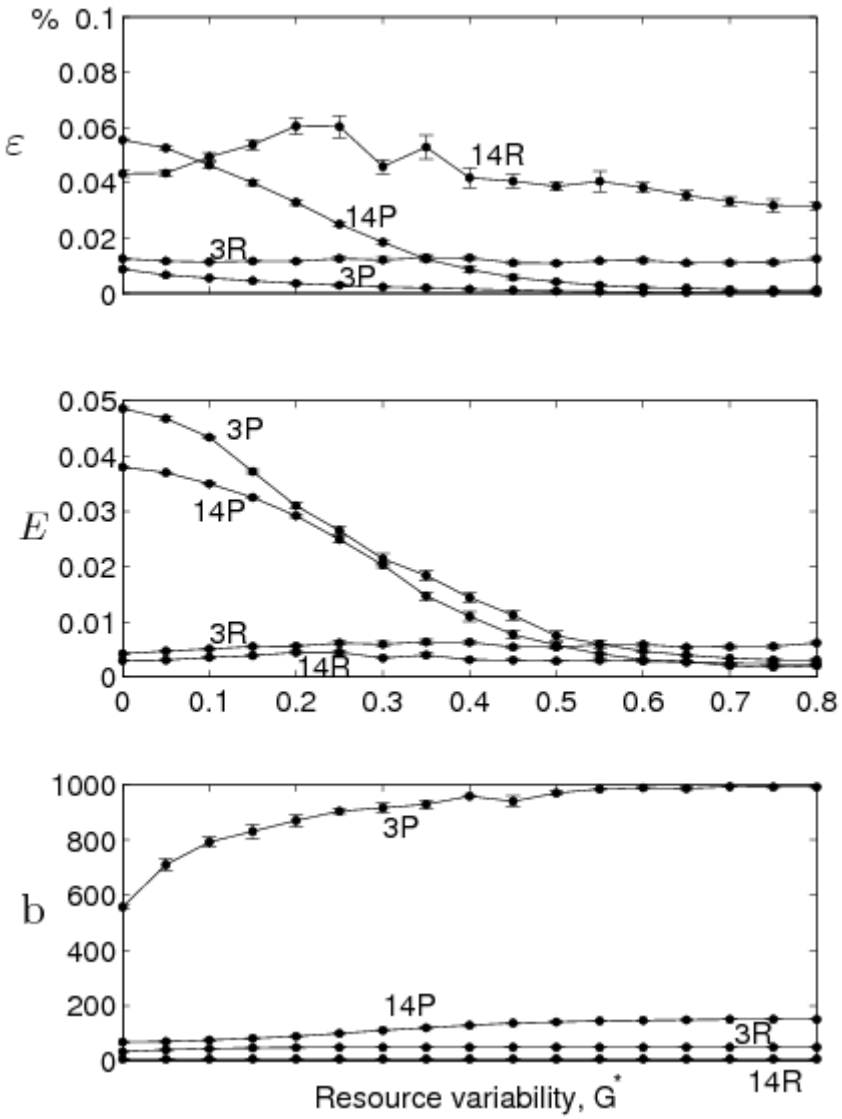
% 0.1
0.08
0.06
0.04
0.02
0
ε
14R
14P
3R
3P
0.05
0.04
0.03
0.02
0.01
0
E
3P
14P
3R
14R
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1000
800
600
400
200
0
b
3P
14P
3R
14R
Resource variability, G*

Figure 5

APPENDIX A

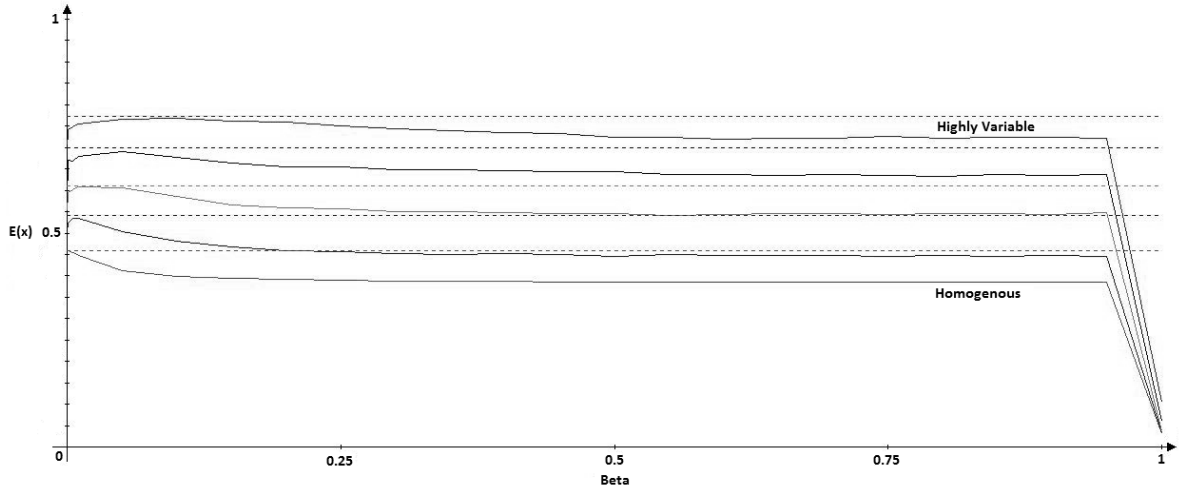

Figure 6. *Landscape-scale mean foraging performance per bee per foraging bout (E(x)) per trial with varying β and varying landscape reward variability. The figure shows the results obtained with a landscape size of three fields and bees equipped with fixed nectar carrying capacities of 500 nectar units.*

It can be seen that β = 0.05 represents a 'good' value in terms of yielding close-to-omniscient foraging performance for the scenario shown. This also holds true for all other tested scenarios. β = 1 (which represents a first-order Markov transition rule) can be seen to be significantly sub-optimal in both cases, highlighting the importance of memory past the last encounter for efficient foraging.

Chapter Four : Recreation of Iconic Patterns of Bee-Mediated Gene Flow by HARVEST

4.1 Introduction

Bumble bees are significant pollinators in many temperate ecosystems in Europe, North America and Asia (Goulson, 2003). They have therefore been widely studied both for their own foraging behaviours (e.g. Heinrich, 1979) and their contribution to the creation of pollen dispersal patterns that influence gene flow levels among plants (Schmitt, 1983). A number of recurrent patterns relating to bumble bee-mediated gene flow have been shown to emerge in plant populations, which have been referred to as *iconic patterns* (Cresswell, 2006). In particular, three iconic patterns have been identified that emerge from generic and repeatable behavioural phenomena exhibited by bees (Cresswell, 2006). First, the level of pollinator-mediated gene flow decreases rapidly as the intervening distance from a pollen source to recipient plants is increased (Crane and Mather, 1943; Cresswell, 2006). Second, the level of pollinator-mediated gene flow from source to recipients decreases with increasing number of flowers in the recipient patch (Klinger *et al*, 2002; Cresswell, 2006). Third, the male success of individual plants varies within plant populations (Devlin and Ellstrand, 1990; Cresswell, 2006). The foraging model can be tested in part by its ability to recreate *in silico* these iconic patterns, thereby beginning to validate the behavioural algorithms I have implemented.

The first iconic pattern requires a spatially-explicit landscape structure, and in the HARVEST model the inter-patch distances are explicitly defined. We can therefore assess the impact of varying inter-patch distances upon residence and traffic levels and determine resultant changes in gene flow. For the second iconic pattern, patches must be distinguished not only by the proportion of flowers containing nectar, but also by the number of flowers in the patch, because individual flowers within patches are explicitly modelled. I therefore assess the impact of varying the number of flowers in a patch upon traffic, residence and gene flow.

In its current state, the model is not able to address the third iconic pattern. Plants are not explicitly modelled, but rather patches are considered to be collections of flowers. Therefore, we cannot assess plant-level success such as those described by the third iconic pattern. Furthermore, the flowers are, in effect, featureless and indistinguishable to bees. In contrast, one of the main reasons that plants vary in reproductive success is because of variation in the architecture of flowers (Galen and Stanton, 1989; Galen and Cuba, 2001; Thomson 1988

English-Loeb and Karban, 1992; Firmage and Cole, 1988; Schoen and Clegg, 1985). Although feasible in principle, all flowers in the model are identical, and so the model is not set up to look at questions such as these. Therefore, I focus only on the first two iconic patterns.

It is useful to assess the model's ability to qualitatively replicate iconic patterns of bee-mediated gene flow dynamics, because we want to justify the model as an accurate predictor of bumble bee behaviour, especially at scales beyond feasible measurement. By showing that the model is able to replicate these patterns, increased credibility is given to the patch choice rules within the model. Additionally, both of the iconic patterns I am investigating are related to bumble bee-mediated gene flow patterns, the prediction of which is my ultimate aim. As such, support from gene flow-related patterns helps to justify the model as a useful tool for predicting pollinator-mediated gene flow generally.

In this chapter, I outline a number of experiments that were conducted *in silico* to assess the ability of the model to replicate the first two iconic patterns. I analyse the efficacy of the model in replicating these results under a range of scenarios and identify additional scenarios where new patterns of gene flow emerge. Furthermore, I explain the reasons why these patterns emerge by exploring the underlying behavioural dynamics that led to such patterns, which is important because the mechanisms that produce the iconic patterns in real landscapes are only partially understood. Therefore, I also offer new insights into the basis of the iconic patterns observed empirically by having the ability to view the bee behaviours that caused the model to replicate these patterns.

The chapter has the following structure. First, I report how a fundamental parameter of the model was optimized in the context of the required landscape. Then, I investigate HARVEST's ability to replicate the two iconic patterns in gene flow among plant populations that I outlined above: the effect of spatial separation and the effect of patch size. Each of these latter sections is presented in two parts (methods and results), before overall conclusions are discussed.

4.2 Investigation (i): Patch scale optimisation of the 'Memory Parameter', β

4.2.1 Introduction to Investigation (i)

Here, β determines the importance placed upon single flower encounters and thereby determines the sensitivity of the bee to its most recent sampling experiences, with β values closer to 1

effectively making bees more responsive by increasing the likelihood of leaving a patch when a run of poor flowers is encountered. Unlike the majority of other parameters in the model, whose values can be estimated by direct measurement, β relates to an abstract internal property of bees. There is unequivocal experimental evidence that the foraging behaviour of bees is influenced by immediate sampling experience (Dukas and Real, 1993; Dall *et al*, 2005; Harder and Real, 1987; Cartar, 1991; Keasar *et al*, 2002), so it is reasonable to postulate a memory span (ie $\beta > 0$), though there is no available measurement of this. However, since β is so crucial to the learning mechanism in the model, we cannot set its value arbitrarily.

I therefore conducted a β-optimisation study in which β values were tested at high resolution over a wide range of values, where I quantified for each the mean foraging performance. Performance, or rate of reward uptake, is a suitable index for determining appropriate β values here, assuming that bee foraging is economically optimised. I conducted two separate β-optimisation studies, and I explain later why a second study was needed. The first β-optimisation study was conducted in a landscape of small patches.

4.2.2 Methods

β values were tested from 0.0001 to 1.0 inclusive in intervals of 0.005 for values from 0.01 to 1. However, I generated additional resolution at the very low values by analysing $\beta = 0.0001$, 0.0005, 0.001 and 0.005. Particular focus was placed on the lower values, because the bees will typically encounter a large number of flowers with variable resources, and theoretically they will require the averaging of a large number of samples to build up a reliable picture of a patch, because of the possible influence of short, stochastic, 'runs of bad luck'. For generality, I also tested these values of β for a variety of different foraging landscapes, because we would ideally like to identify a single value for β that confers generally high performance in any set of circumstances. Specifically, I tested landscapes containing 6 patches, 12 patches and 30 patches, and within each of these combinations I tested four different levels of economic richness by varying floral replenishment intervals. I split the available patches into two types – *rich* and *poor* (i.e. those that replenished more frequently and those that replenished less frequently) – in order to test the effect of variation in patch quality. For this study I tested 1) landscapes in which all patches replenished at the same rate, 2) landscapes in which 'poor' patches took twice as long to replenish as other patches, 3) landscapes in which 'poor' patches took three times as long to replenish as other patches and 4) landscapes in which 'poor' patches took four times as long to replenish as other patches. Rich patches were always set to replenish

empty flowers after an interval of 10 minutes, which equates to 200 flower visits when assuming a bee's mean flower handling time is 3 seconds (Cresswell, 1999).

4.2.3 Results

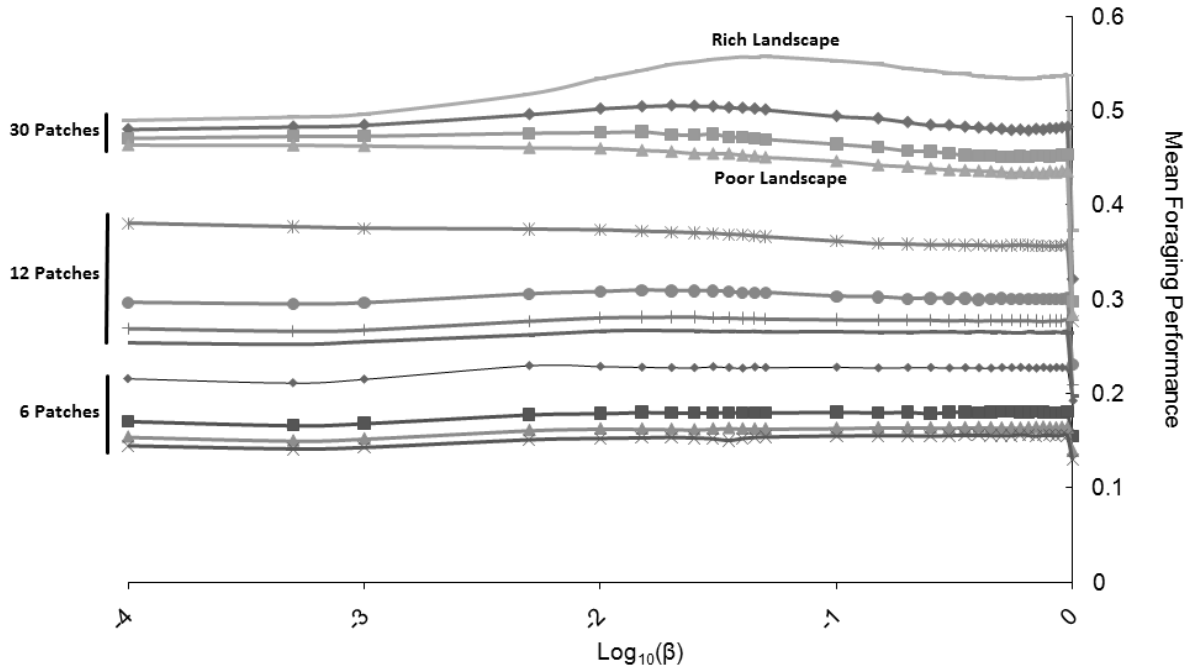

Figure 4.1. Mean foraging performance per bee per foraging bout per trial (y-axis) in small-scale landscapes, with varying β *(x-axis), varying landscape size (number of patches) (annotated plots) and varying economic richness (annotated plots). The figure shows the results obtained from the patch-scale β optimisation study across all of the* β *values and scenarios tested. 14 bees foraging simultaneously; nectar carrying capacity of 10,000 nectar units; 140 flowers per patch; 1 time unit inter-patch flight time between nearest neighbours; 8 hours simulated foraging time per trial; nest 1 time unit from all patches.*

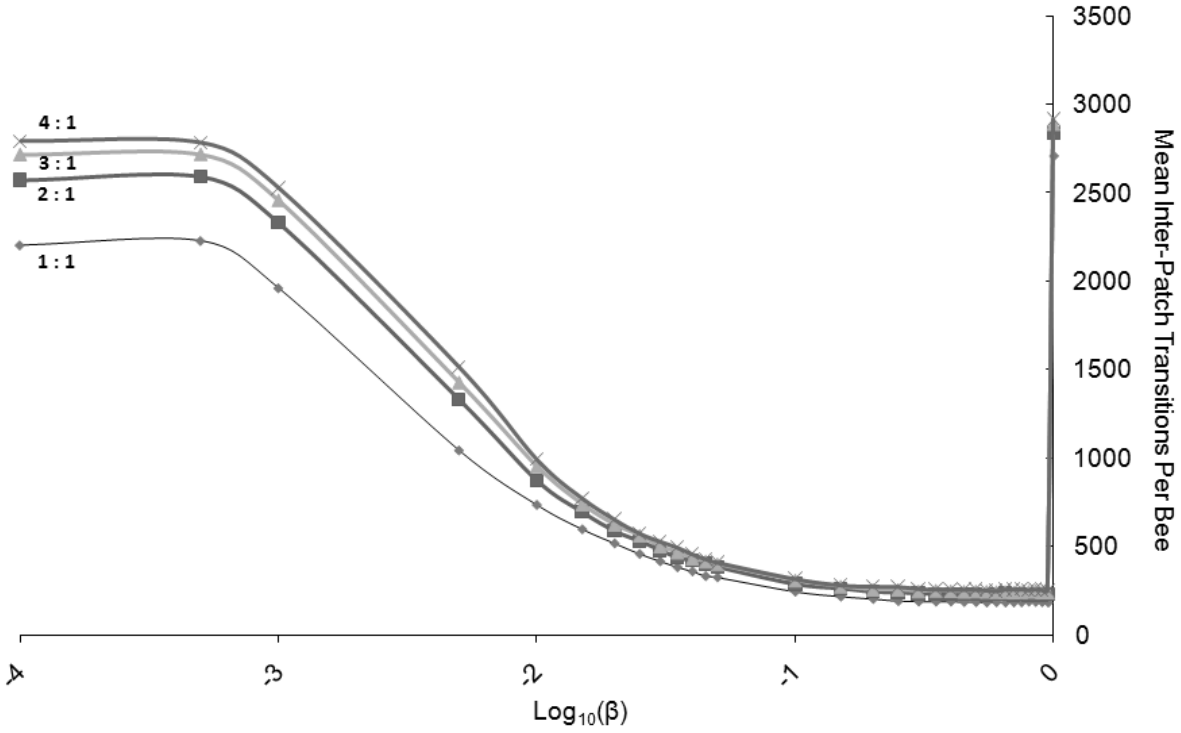

Figure 4.2. Mean inter-patch traffic per bee per foraging bout per trial (y-axis) in a landscape with six patches, varying β (x-axis) and varying economic richness (1 : 1 rich : poor replenishment frequency ratio, up to 4 : 1). 14 bees foraging simultaneously; nectar carrying capacity of 10,000 nectar units; 140 flowers per patch; 1 time unit inter-patch flight time between nearest neighbours; 8 hours simulated foraging time per trial; nest 1 time unit from all patches.

Perforance tends to be largely independent of β values, provided the values are neither too small nor too large (figure 4.1). Otherwise, this suggests that varying β has only a trivial impact upon foraging performance at the small-patch scale. The level of inter-patch traffic shows a general decrease as β increases (figure 4.2), except when β = 1, which yields very high traffic levels.

4.2.4 Discussion

Several conclusions should be drawn from these results. First, when β = 1, there is a drop in performance for all scenarios tested. With β = 1, a bee's estimates of patch quality can only ever be either 1.0 or 0.0, because only the most recently sampled flower contributes to the calculated estimate, which effectively converts the set of patch quality estimates into binary form. The consequence is that whilst the bee in the current patch is encountering full flowers it

will not move, because the action-values of all moves to alternative patches will include a travel penalty, and with the current patch estimate at maximum quality, the action-value representing staying in the patch cannot be improved upon. However, as soon as the bee encounters an empty flower, the patch quality estimate is immediately changed to zero, which will result in one of two behaviours; either there will exist at least one patch in the landscape that the bee has not yet visited with an estimate still set to the default of 0.5, causing the bee to move to this patch, or all patch estimates will be effectively zero, which will cause a random choice of patch. Setting $\beta = 1$ therefore leads to extremely high frequency of travel (figure 4.2) that incurs economic penalties without prospect of improved rewards, which accounts for its relatively poor performance (figure 4.1).

Foraging performance increases with the number of patches in the landscape, and decreases with increasing variability among patch replenishment rates (figure 4.1). Furthermore, these effects remain consistent across all β values tested. This is explained as follows. With the number of bees set at 14, an increased number of patches imply an increased flower availability per bee, which increases overall profitability because it reduces the impact of the bees' depletion effects upon nectar standing crop levels and thereby induces less inter-patch traffic, as bees typically encounter fewer empty flowers. In a landscape with more variable patch replenishment rates, the greater distinction between rich and poor patches encourages bees to increasingly specialise on rich patches. Consequently, nectar standing crop levels across the landscape are homogenised, because bees over-harvest superior patches, whilst proportionally spending less time foraging in patches that replenish infrequently. In a homogenous patch quality landscape, the frequency of inter-patch travel is increased, because patch quality estimates remain similar, and only small fluctuations in these estimates are needed to induce patch departures, which may arise stochastically through 'runs of bad luck'.

In landscapes with few patches, bees' economic performance is generally independent of β for intermediate values. However, varying β does have a pronounced impact upon the frequency of inter-patch transitions, or traffic levels, with higher β values generally leading to reduced traffic levels (figure 4.2), with the exception of when $\beta = 1$. Initially this would appear to be counter-intuitive; higher β implies higher sensitivity to current experience as more emphasis is placed upon individual flower encounters, which would seem to imply an increased potential for instigating inter-patch movements. However, by analysing the estimated qualities over the course of a trial we find the answers to this seemingly counter-intuitive behaviour.

Very high sensitivity to recent binary rewards, which by definition have values of 0 or 1, means that the estimated quality for any particular patch must be close to zero or one. Once the forager has 'learned' a landscape by visiting all the patches, the estimated qualities of vacated patches all tend to be close to zero, with little room for the current patch quality estimate to fall low enough to cause the action-value for staying in the current patch to become worse than an alternative (and instigate inter-patch movement), unless consecutive empty flowers are encountered. If, instead, β is smaller, the estimated qualities of vacated patches are derived from an average over several to many flowers, and so estimates are typically greater, all else being equal. It is, therefore, more likely that the estimated quality of the current patch can fall low enough to make the action-value for staying in the current patch worse than an alternative. Hence less traffic is observed when $\beta \approx 1$, all else being equal. When $\beta = 1$, much higher traffic levels are observed, because patch quality estimates become binary, and patch departures can be instigated as soon as a single empty flower is sampled.

I have now shown that at the small-patch scale foraging performance is largely unaffected by β variation but the overall level of inter-patch traffic is strongly affected. Together these findings imply that increases in traffic at the small-patch scale do not greatly affect foraging performance. This is reasonable, because in the small-scale landscape inter-patch travel incurs only minor time penalties and therefore frequent travel is negligibly costly. However, by the same logic, varying β at the landscape-scale probably will have an impact on foraging performance, as travel is costly due to the increased inter-patch distances, and it would be economically preferable to avoid excessive travel for reasons of rate maximisation and to minimise 'wear and tear' effects (Cartar, 1992; Rodd *et al*, 1980). Therefore, I conducted a second β-optimisation study using large-scale landscapes.

4.3 Investigation (ii): Landscape-scale optimisation of the 'Memory Parameter', β

4.3.1 Methods

I tested a variety of β values across a number of different landscape configurations; specifically 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1 and all other higher β values in intervals of 0.05 up to and including 1.0. To avoid overfitting the model's optimisation to a particular parameterisation, I again tested β against a selection of different landscape scenarios in an attempt to obtain a β value that had a universal optimality. Specifically, I tested landscapes with different levels of variability in patch quality, different numbers of patches, and different carrying capacities of bees (which as I discussed in Chapter Three can be used as a proxy for nectar reward richness).

I expressed patch quality variability in terms of the range between the lowest and highest patch qualities in the landscape, with the highest and lowest qualities assigned symmetrically around a central patch quality of 0.5, and all other patch qualities drawn randomly within these boundaries. I tested ranges in patch quality of 0 (representing a completely homogenous landscape) up to 0.8 (representing a highly variable landscape) in intervals of 0.2. A range of 1.0 was not tested as such extreme variability is unlikely to be found in real landscapes of flowers, because the plants within patches tend to vary in their nectar provision (Pleasants and Zimmerman, 1979; Pleasants and Zimmerman, 1983). I tested landscapes containing either 3, 7 or 14 fields and provided bees with nectar capacities of 125, 500 or 1,000 nectar units.

Each combination of parameter values that I tested was repeated over 10 trials and I took averages over these trial sets. Each trial simulated 8 hours (9,600 time units) of foraging activity. Each patch in the landscape represented a field, and each field contained 60,000 flowers in total. When there was no variability in the landscape, all fields had qualities of 0.5, and so contained 30,000 full flowers and 30,000 empty flowers. To mitigate against potential behavioural effects that could emerge as a result of specific spatial configurations of fields being used, I set all patches and the nest to be equidistant from every other patch. The constant inter-patch distance was 1km, or 45 time units. I allowed 100 bees to forage simultaneously in each trial to improve accuracy in the obtained results. I also repeated trials with omniscient bees which, as I described in Chapter Two, are equivalent foragers that are always aware of the true qualities of all patches. I used the data obtained from omniscient bee trials to contextualise the predicted foraging efficiency of the learning bees.

4.3.2 Results

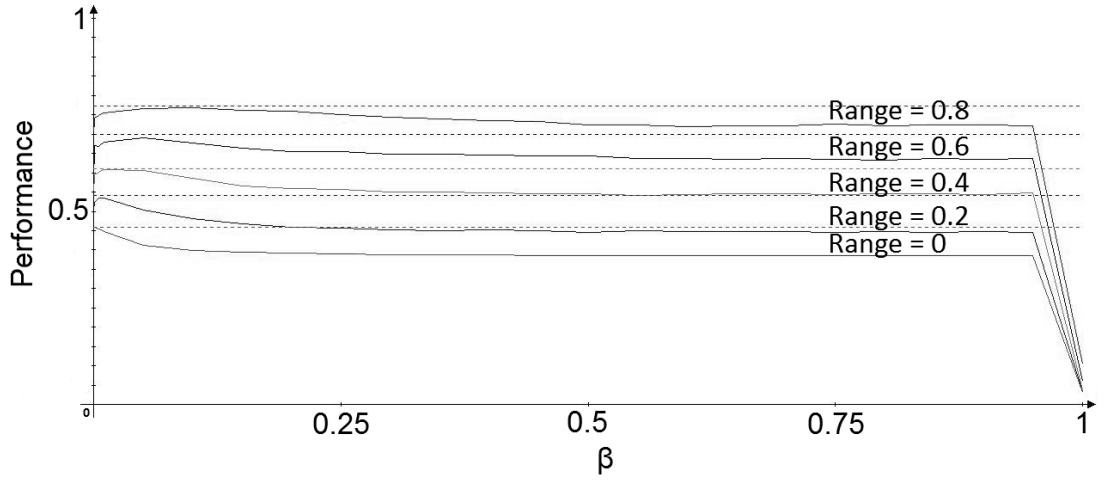

Figure 4.3. Landscape-scale mean foraging performance per bee per foraging bout (P(x)) per trial (y-axis) with varying β (x-axis) and varying landscape reward variability. The figure shows the results obtained with a landscape size of three fields and bees equipped with fixed nectar carrying capacities of 500 nectar units. Dotted lines indicate equivalent omniscient bee performance.

Peak performance occurs when the value of β is less than 0.25 (figure 4.3), and this is consistent for all scenarios tested. In all scenarios, a β value of 1.0 yielded led to extremely inefficient foraging (figure 4.3), as bees instigated an inter-patch move every time they encountered an empty flower. A β value of approximately 0.05 tends to yield strong foraging efficiency across the landscape-scale scenarios tested.

4.3.3 Discussion

When bees are equipped with lower β values, their sensitivity to individual flower encounters is lessened, and the probability of an inter-patch move being instigated is therefore lower for any given flower encounter. Therefore, inter-patch traffic levels are observed to be lower in these cases, which means that bees are spending more time actively foraging rather than travelling. The peak in performance with lower β is more pronounced with larger landscapes, because if a bee is faced with a greater number of foraging site alternatives, the likelihood of them settling in a patch is reduced because the chance of another patch being considered preferable is increased. If this effect combines with increased sensitivity, the result is inevitably lower performance on average for bees that are highly sensitive to new experience in large landscapes.

The results imply that a β value of approximately 0.05 would be an appropriate parameterisation for the model. This is reasonable as we would expect an optimum performance to be found with low individual reward sensitivity, because the fields contain extremely large numbers of flowers, and therefore estimates of patch quality based on smaller numbers of flower samples are more likely to be unrepresentative and lead to erroneous inter-patch movements. The opportunity cost of these errors is largest when patch separation is greatest.

4.4 Investigation (iii): Iconic Pattern 1 : Increasing inter-patch distance decreases gene flow levels

4.4.1 Introduction to Investigation (iii)

Empirical studies have investigated the influence of distance upon gene flow levels mediated by pollinators (Bateman, 1947; Damgaard and Kjellson, 2005; Fenster, 1991, Cresswell, 2006). This is an important avenue of investigation, because an obvious solution to minimising the threat of cross-pollination to gene confinement is to increase the separation distance between plant populations and thereby minimise the probability of pollinators making inter-patch movements (Ramsay *et al*, 2003; Colbach *et al*, 2009; Llewellyn *et al*, 2007). Investigations have looked at the impact of separation distance upon gene flow both at smaller scales (e.g. Crane and Mather, 1943) and larger kilometre scales (e.g. Rieger *et al*, 2002). These studies assess gene flow levels by quantifying the presence of marker genes that originate in the source population, and analysing the amount of marked progeny present within unmarked recipient populations at varying distances from the genetically marked source.

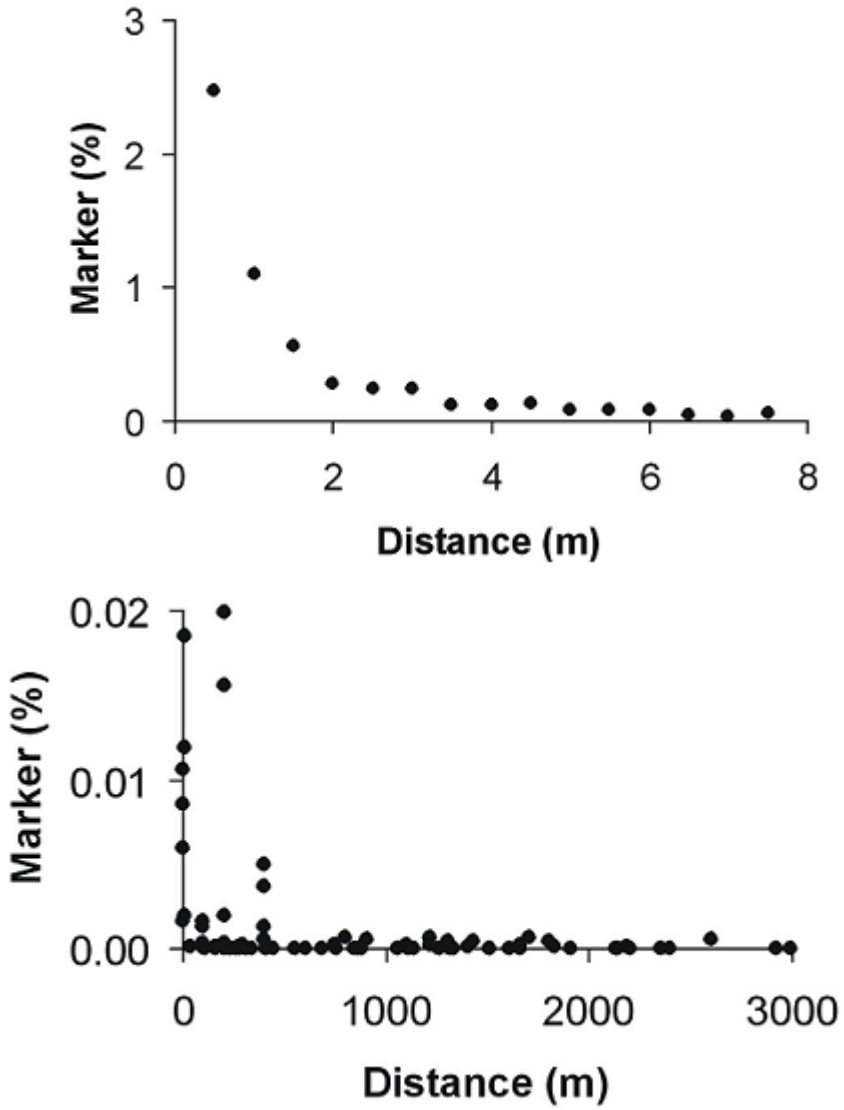

Figure 4.4. Examples of the iconic gene-flow distance relationship. These data sets were obtained by assessing the percentage of genetically marked seed set on marked plants (y-axis) with increasing distance between populations (x-axis). A decelerating decrease in gene flow levels, present in both examples here, is a common observation with such experiments at both small scales (upper panel) and larger scales (lower panel) (figure from Cresswell, 2006).

The results of these investigations consistently show that the proportion of marked progeny in the recipient populations shows a leptokurtic decline with increasing separation distance between the sink population and the marked source (figure 4.4). There are a number of reasons why this distance-related pattern emerges. First, whilst bumble bees are in transit between marked and unmarked patches, pollen may be dislodged by self-grooming processes (Rademaker *et al*, 1997) or by the effect of wind (Harder and Barrett, 2006). Over longer distances, the risks of pollen loss will be greater because the original amount of pollen brought

from the source patch would be further depleted if the journey time was greater. In addition, due to the extremely patchy nature of most real-world foraging landscapes (Ricketts, 2001; Westphal *et al*, 2006; Kohlmann and Risenhoover, 1998; Klaassen *et al*, 2006; Iwasa *et al*, 1981), bees potentially stop to forage at small patches of heterospecific / conspecific plants in the intervening landscape between the marked and unmarked patches (Ricketts, 2001). In doing so, bees would deposit and begin to exhaust marked pollen from the source population. Over larger distances, assuming there is an even distribution of small patches in the landscape, the probability of finding and stopping at such smaller patches would be increased, and this could explain why the representation of marked progeny is lower when the sink population is a greater distance from the source.

In contrast, the model assumes that inter-patch flight is an uneventful activity for pollen carried by a bee, and that it is effectively a state of transit for the bee until it arrives at its destination, because we do not explicitly track pollen within the model; we assume its unimpeded transmission once the bee movements are solved. Therefore, the model makes no allowance for the influence of pollen being dislodged during transit. Similarly, the landscapes I simulate are simple, comprising only large patches separated by bare intervals. Thus the gene flow levels between a pair of patches are based on direct moves from source to sink with no 'absorption' at any other patch. In principle, HARVEST is capable of simulating both pollen depletion in transit and the effects of landscapes with very small clumps of flowers separating larger patches of flowers, but it is not the focus of this study because (a) there is little or no empirical data to provide a basis for modelling either effect and (b) we want to determine whether the effects of pollinator movements alone can govern the emerging patterns.

It is important to be clear when talking about patches being 'more distant', as this can be in one of two ways. First, patches may become more isolated from not only the source patch, but from all other patches in the landscape. This can be conceptualised abstractly as taking a sink patch and moving it away from the source patch, but with the condition that it does not move closer to any other patches. This latter condition is unlikely to be biologically realistic, but it is possible to imagine such scenarios theoretically. The more likely scenario in the real world involves flowers distributed in a patchwork, where more distant patches from a particular source are also necessarily closer to patches other than that source patch. At first glance, this may not seem like an important distinction to make, but since the model makes the reasonable assumption that bees consider flight distances when choosing amongst patches, the presence of nearby non-source patches could attract bee foraging activity and thereby alter the residence and the traffic into and out of the sink patch, which in turn would influence gene flow levels. I look at both

types of scenario, which I refer to as the 'simple separation' scenario (involving two patches) and the 'complex catchment' scenario (involving multiple patches).

4.4.2 Methods

4.4.2.1 Patch Classifications

The patches in each experiment can be categorised either as *source* patches, *sink* patches or *catchment* patches. A source patch represents the population from which gene flow occurs, and only a single patch is nominated as a source patch. A sink patch is a population to which gene flow from the source can occur. Catchment patches are any patches that are considered to be neither source nor sink, instead representing 'background' patches that may influence foraging behaviour.

4.4.2.2 The 'Spatially-Explicit' Landscape

I assume that bees fly at a speed of 7 m s^{-1} when travelling between fields (Riley *et al*, 1999), and so a distance of 1km would be covered by one of the model bees in 45 time units. The landscape in the model is designed to be spatially-explicit with respect to the placement of resource patches within a landscape, but within patches the locations of flowers are not spatially explicit. Each patch can be considered as a 'bag' of resources from which flowers are randomly 'drawn' when a bee samples a flower in that patch. While the size of a patch in the model determines the total number of flowers available within that patch, it does not determine the space that the patch occupies within the foraging landscape. For simplicity, all patches within a landscape are assumed to be points, and within-patch flights by bees are included in the flower handling time.

The bees' nest has no spatially explicit position in the landscape, but is equidistant and trivially distant from all patches in the landscape. I do this to remove any possible interference from placing the nest in a specific location, and to focus my initial investigation solely on the effect of inter-patch distance on gene flow. This simplification also reflects a real-world phenomenon, namely that bumble bee nests are notoriously difficult to locate (Suzuki *et al*, 2007; Dramstad, 1996) and so accurate placement of the nest in many simulated scenarios would be impossible. Therefore we do not have enough knowledge to create a more realistic entry for bees into our simulated landscapes.

All patches in the experiments begin with half of their flowers full and half empty ($Q(i)$ = 0.5 for all i). When simulating at the small-patch scale, depletion and replenishment effects are included to emulate the impact on standing crop levels that bees have on patches with relatively few flowers. In small patches, foragers could reduce standing crop levels and experience patch depletion by their harvesting activities. I therefore simulate depletion and replenishment effects by allowing empty flowers to replenish (become full) after 10 minutes. The interval of 10 minutes is derived from investigations of bumble bee traplining, which suggest that bees sometimes re-visit flowers as frequently as once every 6 to 20 minutes (Williams and Thomson, 1998). Assuming that bees are harvesting profitably and systematically, then most flowers must have replenished to an acceptable level in this time. Presumably, this relatively rapid replenishment represents a relatively rich landscape for the bees.

When simulating at the larger landscape-scale, where resource patches equate to agricultural fields of mass-flowering crop, I simulated 100,000 flowers per field and removed depletion and replenishment effects since, in reality, bees are sparse in these fields (Hoyle *et al*, 2007) and therefore unlikely to revisit a flower – only a small proportion of flowers are visited by bees at this scale (Cresswell *et al*, 2002). Therefore, this approach replicates the trivial impact of bees on standing crop levels of nectar and pollen in larger foraging sites.

4.4.2.3 Solving the E-Psi-b model

To solve the E-Psi-b model of gene flow, I analyse the transition matrix (which provides information about the relative frequencies of inter-patch moves) to calculate the levels of bee-mediated gene flow from the source patch to the sink patch with varying distance from source to sink. The proportion of moves made by bees from the source patch to the sink patch in the transition matrix allows us to determine E within the gene flow model. In addition, the residence results we obtain allow us to determine b, which represents the mean number of flowers visited by bees in the sink patch. As stated in Chapter Two, ψ is set to be a constant of 1 throughout this study of bumble bees foraging on *B. napus*, as this is based on empirical investigations.

4.4.2.4 General Parameterisations (Consistent for all Experiments)

In each trial, seven bees forage simultaneously in the landscape; this abundance of bees was chosen to match observations from the small scale empirical study reported in Chapter Five. At the start of each trial, all bees set the estimated quality of each patch to be $q(i)$ = 0.5. The bees

forage for 8 hours (9,600 time units) because bees generally forage most actively between around 0900 and 1700 (Cresswell, personal observation) and empirical bumble bee studies also tend to make observations within these times (e.g. Osborne *et al*, 1999; Hodges, 1985a; Plowright and Galen, 1985). In each experimental trial, average behaviour was taken over 20 replicates to characterise variability.

Each bee can carry 500 units of nectar. The minimum amount of nectar that a bee must extract from a flower in order for the sampling transaction to be profitable in energetic terms is related to the sugar present in the nectar (Cresswell *et al*, 2000). Cresswell *et al* (2000) showed that if a nectar sample is assumed to have a typical sugar concentration of 60% w/v, the minimum reward per flower required to make harvesting profitable for a bee would be 0.15 µl of nectar. The capacity of a bumble bee's honey stomach is about 80µl (Cresswell *et al*, 2000), which means that a bee needs to visit 80µl / 0.15µl = 533 nectar-bearing flowers. Bumble bees have been observed to visit around 500 flowers during a foraging bout in agricultural fields of B. napus (Cresswell *et al*, 2002). In keeping with these values, I assume that the model bees must harvest 500 non-zero floral rewards to achieve capacity. On doing so, they return to their nest to unload before continuing. To clarify, I also assume that pollen collection is equivalent, with nectar collection in the model essentially used as a proxy for pollen collection.

Parameter	Identifier	Value
B	Number of bees in the grid	7
L	Number of patches in landscape	3 (Experiments IP1.1 and IP1.2) 27 (Experiment IP1.3)
Q	Initial quality of patches	0.5
q(all)	Estimated quality of each patch	0.5
β	Sensitivity to individual flower sample	0.05
C	Bee's nectar carrying capacity	500
h	Flower handling time	1 time unit (= 3 seconds)
t(source, sink)	Flight time between source and sink patches	0.2km to 4.2km (Experiments IP1.1 & IP1.2) 1km to 6.1km (Experiment IP1.3)
F_{all}	Number of flowers in each patch	400 (Experiment IP1.1) 100,000 (Experiments IP1.2 & IP1.3)
I_{all}	Replenishment interval for each patch	10 minutes
R_{full}	Reward (in nectar units) offered by a full flower	1
R_{empty}	Reward (in nectar units) offered by an empty flower	0

Table 4.1. Summary of parameterisation for investigation (iii)

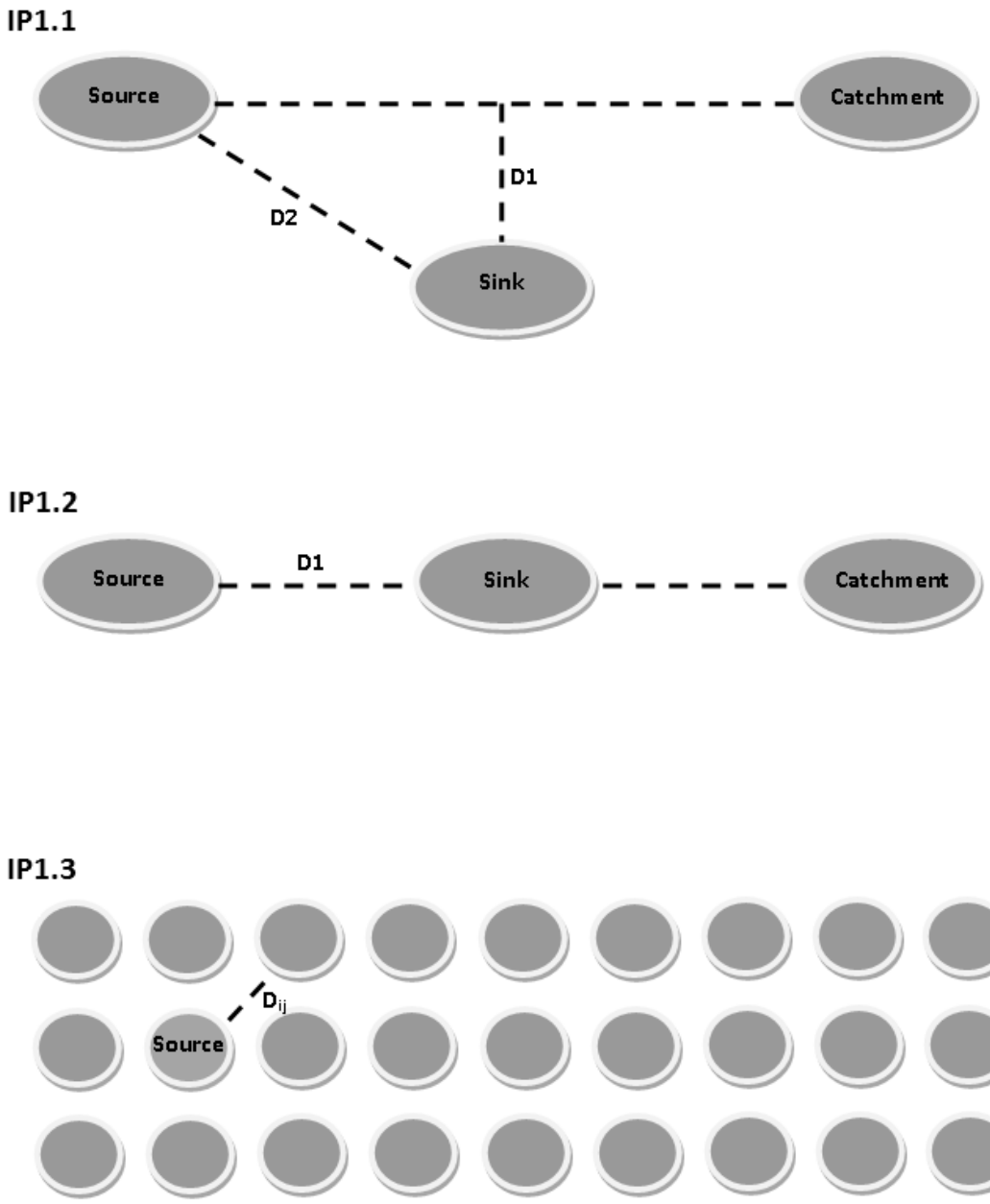

Figure 4.5. Summary of landscape configurations used for investigation (iii). Experiment IP1.1 tests the effect of moving a sink patch further away from the source and all other patches in the landscape, increasing its isolation. Experiment IP1.2 is similar, but the movement of the sink away from the source moves it closer to a third "catchment" patch, resulting in peaked isolation of the sink patch. Experiment IP1.3 considers a matrix of 27 patches, with one embedded source patch, and every other patch in turn considered to be a potential sink.

4.4.2.5 Experiment IP1.1

I first investigate the effect of moving a sink patch further away from a source patch, for the case where it becomes increasingly distant from all patches in the landscape. Consider a

landscape containing three small patches of floral resources, each offering 400 flowers. The small-scale landscape simulated in this experiment mirrors the scale of many of the empirical studies that have identified the iconic pattern in the distance-gene flow relationship (e.g. Crane and Mather, 1943; Devlin and Ellstrand, 1990). One of the three patches in the model landscape is nominated as the source for pollen-mediated gene flow. Another patch is nominated as the sink population and a third patch represents, in simplest form, the context of a surrounding landscape, which I term a 'floral catchment'.

Two of the patches – the source patch and the catchment patch remain in the same location throughout all trials in the experiment. The location of the sink patch is varied by altering distance *D1*, so that the distance between the sink and the source increases in each successive trial, but the equidistant relationship of the sink to the source and catchment is maintained (figure 4.6a).

4.4.2.6 Experiment IP1.2

In the second experiment, I again move the sink patch away from the source, but in doing so the sink patch moves closer to the catchment. Therefore, the sink patch is at its most isolated when equidistant from both the source and the catchment. This is realistic, because patches in real landscapes have numerous neighbours. As such, I implement field-scale patch sizes and, specifically, each patch contains 100,000 flowers.

Once again, the source field and the catchment field remain in a constant location across all of the trials in the experiment. The source field is placed at one end of the foraging landscape, whilst the catchment field is placed at the other end with a distance of around 4.5km (210 time units) separating them. In the first trial set, the sink field is placed directly in between the source field and the catchment field and 10 time units away from the source field (or 200 time units away from the catchment field). In each subsequent trial, the sink field is moved 10 time units further away from the source field and 10 time units closer to the catchment field (figure 4.7a). All other parameterisation of the model remains identical to Experiment IP1.1.

4.4.2.7 Experiment IP1.3

In the third experimental design, I represent more realistic landscapes as a large matrix of fields at the landscape-scale. The previous two experiments purely test the effect of distance upon

gene flow, minimising any possible 'interference' from the presence of a large catchment of patches. However, real foraging landscapes contain many foraging locations scattered throughout the environment and that will influence transfers of pollinators between source and sink fields. In this experiment, the effect of intervening patches can be explored by analysing various pairs of patches chosen from the many in the complex matrix.

The landscape I simulated contained 27 fields set in a regularly spaced matrix with nine columns and three rows. Each field is separated from its nearest neighbours by a distance of 1km, with all other distances calculated geometrically by assuming that a bee's flight path between any two fields is the shortest possible. A single field is designated as the source field, which was selected to not be on a matrix edge. Thus, I designated the field in the third column, second row to be the source field for this experiment (figure 4.8a). Once the trials had all been run, each other field in the landscape was in turn considered to be the sink and was analysed for source to sink gene flow levels based on the foraging results from the trial. All other parameter values remained the same as for experiment IP1.1.

4.4.3 Results

4.4.3.1 Experiment IP1.1

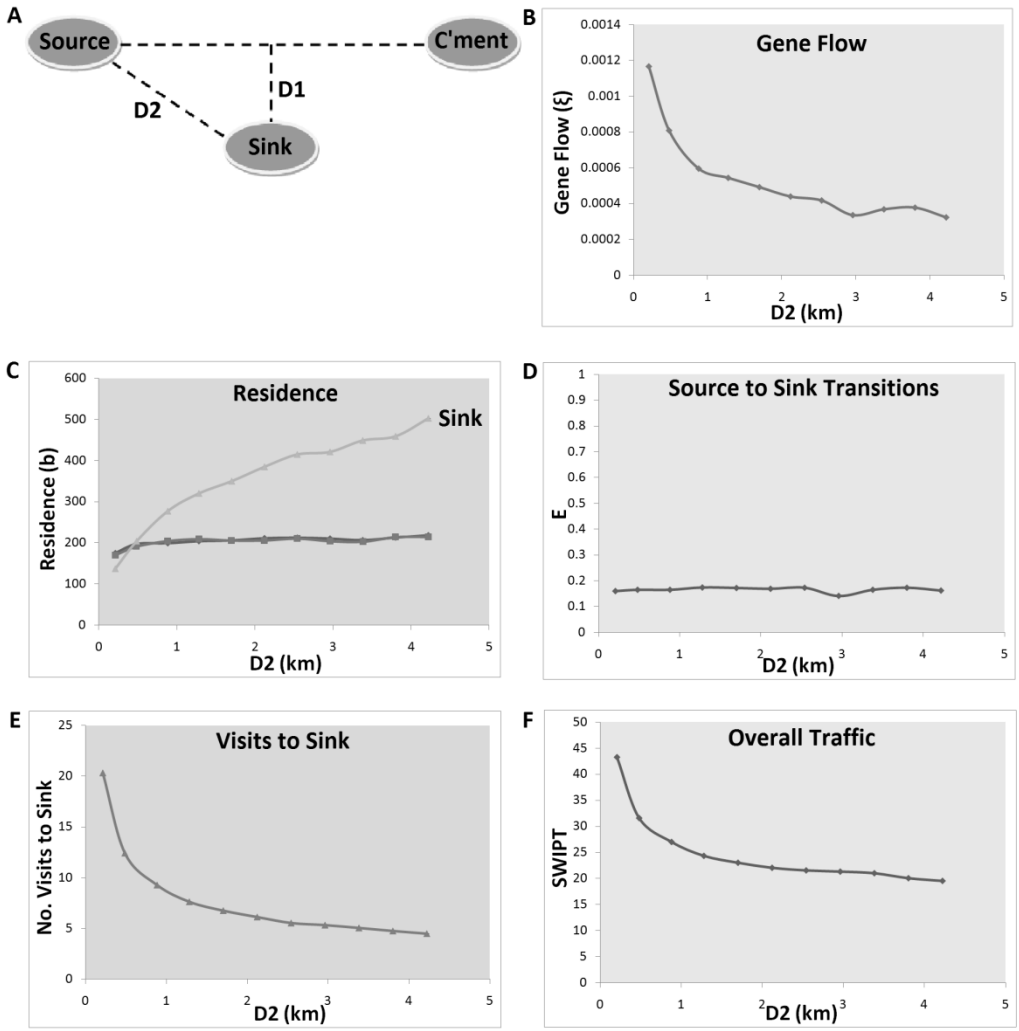

Figure 4.6. Experimental configuration and results for experiment IP1.1 testing for iconic pattern 1. A) The landscape configuration for the experiment. In each successive trial, the sink is moved further along the vertical axis, increasing both D1 (vertical axis displacement) and D2 (distance from source to sink). B) Predicted level of gene flow from the E-Psi-b model (y-axis) with increasing distance from source to sink (x-axis). C) Residence (number of flowers visited) of bees in each of the three patches (y-axis) with increasing distance from source to sink (x-axis). D) The probability of a bee visiting the sink after visiting the source (y-axis) with increasing distance from source to sink (x-axis). E) Number of visits to the sink patch (y-axis) with increasing displacement of the sink from the source and catchment patches (x-axis). F) The level of System-Wide Inter-Patch Traffic (SWIPT – y-axis) with increasing displacement of the sink from the source and catchment patches (x-axis).

The simulation replicates the leptokurtic decay in the gene flow-distance relationship found in empirical studies (figure 4.6b; figure 4.4). According to $\xi_{AB} = \frac{E_A \Psi}{b}$ (equation 2.4 in Chapter Two), decreased residence implies decreased values of *b* and higher gene flow levels, all else being equal. Thus, the gene flow-distance relationship arises, in part, through a mechanism based on variation in residence, which is related to separation distance.

As the sink patch becomes more isolated, bees tend to visit the patch less often (figure 4.6e) because it becomes less economically attractive as an alternative as it becomes more distant, because of travel time. When the bees do visit this patch, they tend to stay longer (figure 4.6c) because it is more isolated, and therefore both alternative patches are only viable alternatives (have higher action values) when the bee experiences a significant run of bad luck (successive empty flowers) in the sink patch to offset the distance penalty required to move to the source or catchment patch.

The probability of a bee making a direct move from source patch to sink patch remains virtually constant regardless of the distance of the sink from the source (figure 4.6d). It might be expected that the probability of moving from source to sink would decrease as the distance of the sink from the source increases. However, we also know that the levels of system wide inter-patch traffic (SWIPT) decrease with increasing distance between source and sink (figure 4.6f). Thus, although the absolute number of moves to the sink decreases with its increasing isolation, as a *proportion* of the total moves made within the system this remains the same. Bees cannot ignore the sink patch and forage exclusively in the source and catchment patches, because eventually these patches would become depleted of resources. The increased cost of travelling to the sink is offset by the decreasing qualities of the source and catchment patches due to depletion. Overall, a neutralising effect emerges and the probability of moving from source to sink remains near constant regardless of landscape configuration.

Therefore, we find that *E* remains effectively constant and the gene flow-distance relationship arises solely from variation in sink residence caused by increased isolation that influences the economic cost of travel. Overall, the results of this experiment suggest that the iconic gene flow-distance relationship emerges simply because the bees spend longer in isolated patches. The relationship is leptokurtic because the relationship between the separation distance (*D2*) and residence is itself non-linear. The non-linearity occurs as bees have a limited fixed nectar carrying capacity and therefore only need to collect a limited number of resources before

finishing the foraging bout. At the highest tested levels of isolation, bees are visiting in the range of 450 – 500 flowers in the sink patch and can therefore fill significantly towards their capacity in the sink field alone, assuming it is not being over-harvested and the patch is maintaining good standing crop levels.

4.4.3.2 Experiment IP1.2

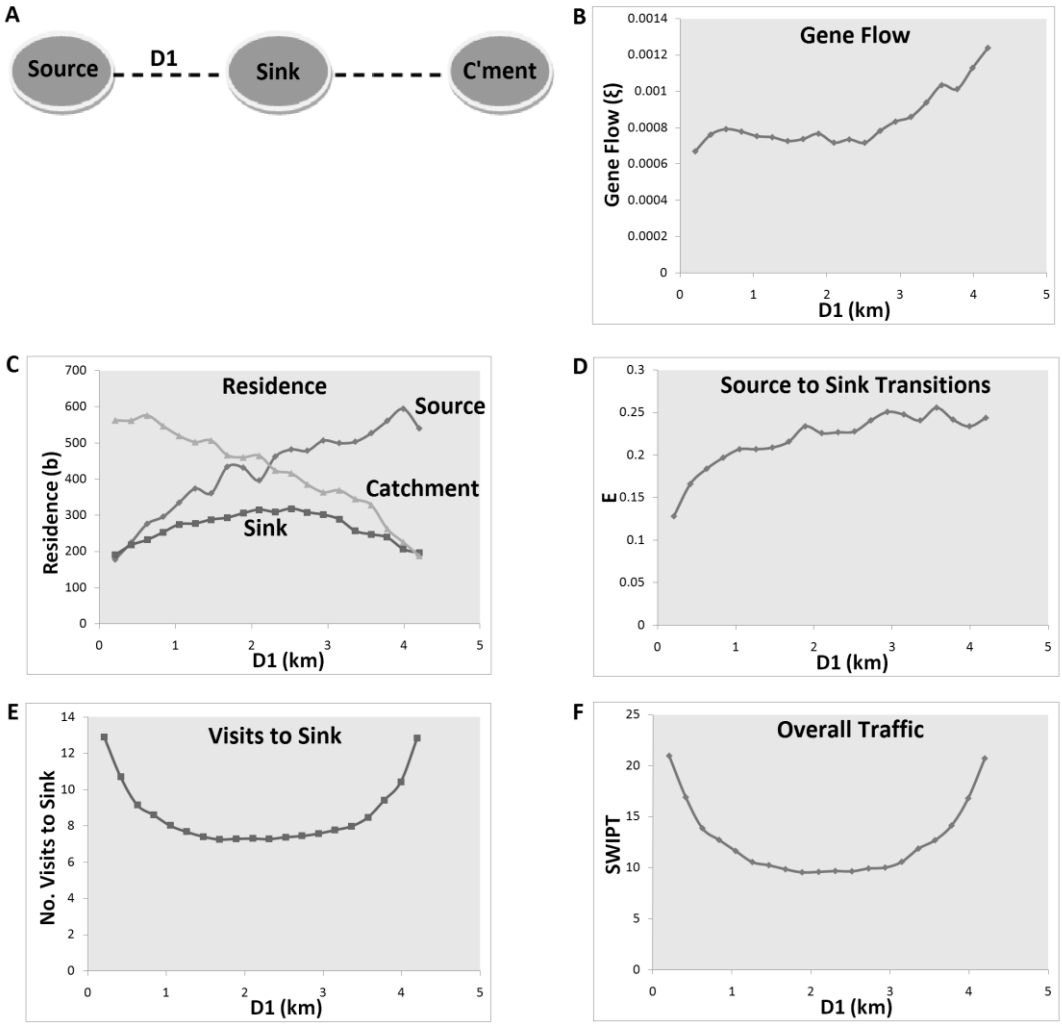

Figure 4.7. Experimental configuration and results for experiment IP1.2 testing for iconic pattern 1. A) The landscape configuration for the experiment. In each successive trial, the sink is moved further along the horizontal axis, further from the source and closer to the catchment, increasing D1 (distance from source to sink). Panels B) – F) are as per Figure 4.6.

We do not observe the iconic gene flow-distance relationship in this experiment, despite configuring a landscape that matches the criteria of increasing the distance of a sink patch from a source patch. The pattern observed here shows that gene flow *increases* as the sink field

becomes increasingly separated from the source field - i.e. when it approaches the catchment field (figure 4.7b). Residence in the sink field peaks when the sink is at its most isolated location (intermediate separation) (figure 4.7c). All else being equal, we know that decreases in residence in a sink population increase gene flow levels into that population. Additionally, the probability of making a direct move from source to sink increases as the sink becomes more distant from the source field (figure 4.7d). Specifically, *E* doubles, and necessarily so does gene flow.

The probability of source-to-sink transition (*E*) is very low when the sink is just 0.2km from the source (figure 4.7d). This is because there are more inter-patch transitions (as bees are moving frequently between source and sink), but since *E* is lower, this represents a smaller absolute number of transitions. This is largely irrelevant, however, as the E-Psi-b model assesses the impact of a pollinator's visit on gene flow levels, not the absolute number of visits made to the sink, but I raise it here to clarify the apparent anomaly.

In principle, the results of this experiment undermine the generality of the iconic pattern and suggest that the principle that simply states "increased distance from source to sink equates to higher pollinator-mediated gene flow" is not adequate to capture the range of possible intricacies of landscape configuration. My experiment here is only slightly altered from experiment IP1.1 and yet the gene flow pattern with varying distance between source and sink has completely changed.

4.4.3.3 Experiment IP1.3

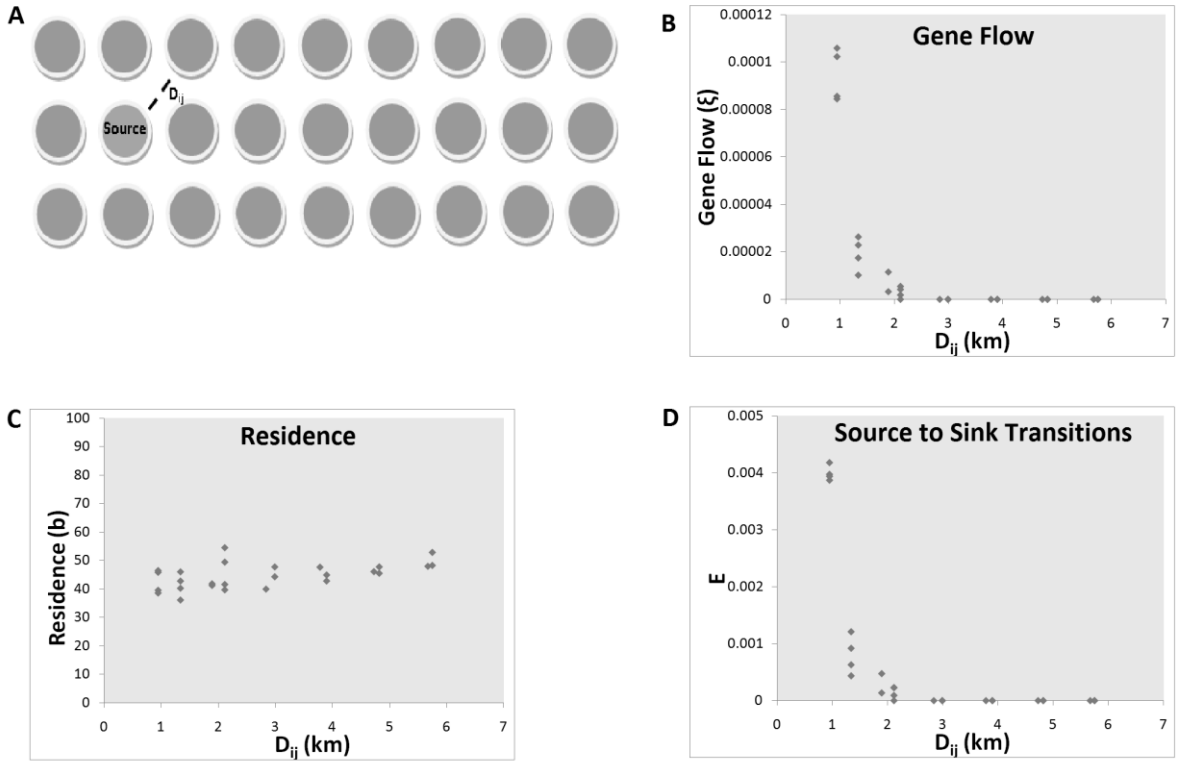

Figure 4.8. Experimental configuration and results for experiment IP1.3 testing for iconic pattern 1. A) The landscape configuration for the experiment. All tested distances are considered simultaneously, by allocating a single source within a matrix of patches. Each non-source patch is considered to be a sink in turn. B) Predicted level of gene flow from the E-Psi-b model (y-axis) with increasing distance from source to sink (x-axis). C) Residence (number of flowers visited) of bees in each of the patches (y-axis) with increasing distance from source to sink (x-axis). D) The probability of a bee visiting the sink after visiting the source (y-axis) with increasing distance from source to sink (x-axis).

As in the first case, the simulated gene flow-distance relationship replicates the iconic pattern (figure 4.8b) – in particular as compared to the landscape-scale example shown in Figure 4.4. This pattern emerges largely from a single cause – the effect of distance on E (figure 4.8d). All else being equal, simulated bees favour visits to nearest neighbour patches because inter-field distances are large – 1km between nearest cardinal neighbours – and movements to a distant field are economically unfavourable. Therefore the majority of moves away from the source field are to nearest neighbours and hence gene flow is highly localised. In this experiment, *E* is the driving force for spatial variation in gene flow pattern that emerges, which contrasts with the dominating influence of *b* in experiment IP1.1. This finding supports the theory that gene flow declines over distance because of bees stopping at intervening patches and therefore exhausting

pollen loads from source flowers. The matrix configuration demonstrates the absorbing power of intervening patches when large, widely separated fields are being considered.

4.5 Investigation (iv): Iconic Pattern 2 : Increasing sink patch size decreases gene flow levels

4.5.1 Introduction to Investigation (iv)

The second commonly observed pattern relating to pollinator-mediated gene flow involves the influence of patch size on gene flow levels. As the number of flowers in a patch increases, we typically observe a decline in the level of gene flow to this patch from an extrinsic source (Figure 4.9; Klinger *et al*, 1992; Goodell *et al*, 1997). Typically the empirical studies that have identified this pattern have focused on small-scale patches because obtaining gene-flow measurements in larger-scale patches would be more difficult. However, the model can simulate patches containing very large numbers of flowers, and I therefore attempt to extend the investigation of this iconic pattern by exploring larger-scale dynamics.

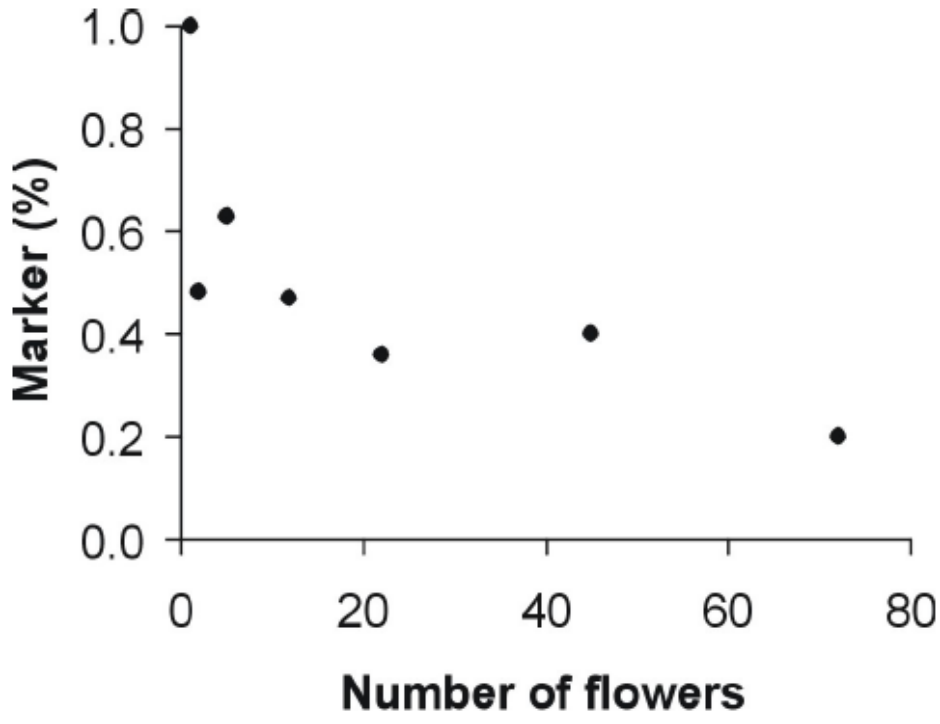

Figure 4.9. An example of the typical pattern of the patch-size gene-flow relationship. As the number of flowers in a sink patch is increased (x-axis), there is a rapidly saturating decrease in the proportion of genetically marked (source) seed observed on the recipient plants (y-axis) (figure from Cresswell, 2006).

The relationship between gene flow and patch-size is perhaps more instantly comprehensible than the relationship between distance and gene flow. The E-Psi-b model (Cresswell *et al*, 2002) tells us that an increase in the pollinator's residence, *b*, in a sink population will have the effect of decreasing gene-flow levels into that population. Larger patches of flowers can retain a pollinator's interest for longer than a patch with fewer flowers, as the chance of encountering a flower that has been depleted of nectar is reduced. Even in large patches, however, bees tend to only visit a relatively small proportion of flowers (Cresswell *et al*, 2002), so the residence-patch size relationship saturates.

It is unclear why bees would exhibit such behaviour because large patches of flowers would seemingly present an attractive opportunity for foraging with minimal cost of travel. Possibly, the effect is specific to particular study systems where bees experienced poor foraging in the flowers they sampled and became convinced that it would be more efficient to forage elsewhere. Alternatively, bees may be pre-programmed by natural selection in some way to sample only a specific number of flowers within a patch, but this is less credible because it would be an extremely inefficient practice at the landscape-scale where inter-patch travel is economically punitive. Perhaps bees differentiate between the small-scale foraging conducted here and landscape-scale foraging in fields? To clarify the basis of this pattern, I tested to see if such behaviour emerged simply from the economics of foraging using the HARVEST model.

It should be noted that simulated bees are not aware of the size of a patch – they experience only the qualities of flowers available within them. Thus, a bee in the model would consider a field containing 50,000 full flowers out of a total of 100,000 flowers to be equally profitable to a patch containing one full flower out of a total of two. Therefore, varying patch sizes will have no direct impact on the bee's internal state. Indirectly, however, an increased number of flowers in a patch decreases the impact of a single bee's foraging on the frequency distribution of standing crop rewards in that patch by reducing the depletion effect, all else being equal. Consecutive visits to empty flowers instigate inter-patch travel, so varying patch size could nevertheless influence foraging behaviour in the model even without the foragers being explicitly aware of the size of a patch. Specifically, bees become increasingly departure-prone as foraging continues in small patches because of encounters with depleted flowers, whereas the probability of departure is effectively constant in large patches. Potentially, this mechanism could generate the residence-patch size relationship required to drive the gene flow-patch size relationship.

4.5.2 Methods

To test the model's ability to replicate the second iconic pattern, I again conducted three separate sets of experiments to explore different landscape configurations and scenarios. Hereafter, when I refer to patch size, I refer to the total number of flowers within a patch. I tested inter-patch distances at both small and large scales, but patch sizes were always smaller in scale, and did not represent large fields of flowers. I therefore model depletion and replenishment effects for all experiments, as patches are small enough for bees to be likely to re-encounter depleted flowers in a single visit to a patch. I explore both simultaneous and consecutive adjustments to patch size and additionally investigate how to manipulate gene flow saturation speeds.

Parameter	Identifier	Value
B	Number of bees in the grid	7 (Experiment IP2.1 & 2.3) 42 (Experiment IP2.2)
L	Number of patches in landscape	3 (Experiment IP2.1 & 2.3) 27 (Experiment IP2.2)
Q	Initial quality of patches	0.5
q(all)	Estimated quality of each patch	0.5
β	Sensitivity to individual flower sample	0.05
C	Bee's nectar carrying capacity	500
h	Flower handling time	1 time unit (= 3 seconds)
t(source, sink)	Flight time between source and sink patches	1km (Experiment IP2.1 & 2.3) Variable (Experiment IP2.2)
F_{source}	Number of flowers in source patch	1,200 (Experiment IP2.1 & 2.3) Variable (Experiments IP2.2)
F_{sink}	Number of flowers in sink patch	100 - 1,200 (Experiment IP2.1 & 2.3) 2 – 1,000 (Experiments IP2.2)
I_{all}	Replenishment interval for each patch	10 minutes (Experiments IP2.1 & 2.2) 2 minutes – 10 minutes (Experiment IP2.3)
R_{full}	Reward (in nectar units) offered by a full flower	1
R_{empty}	Reward (in nectar units) offered by an empty flower	0

Table 4.2. Summary of parameterisation for investigation (iv)

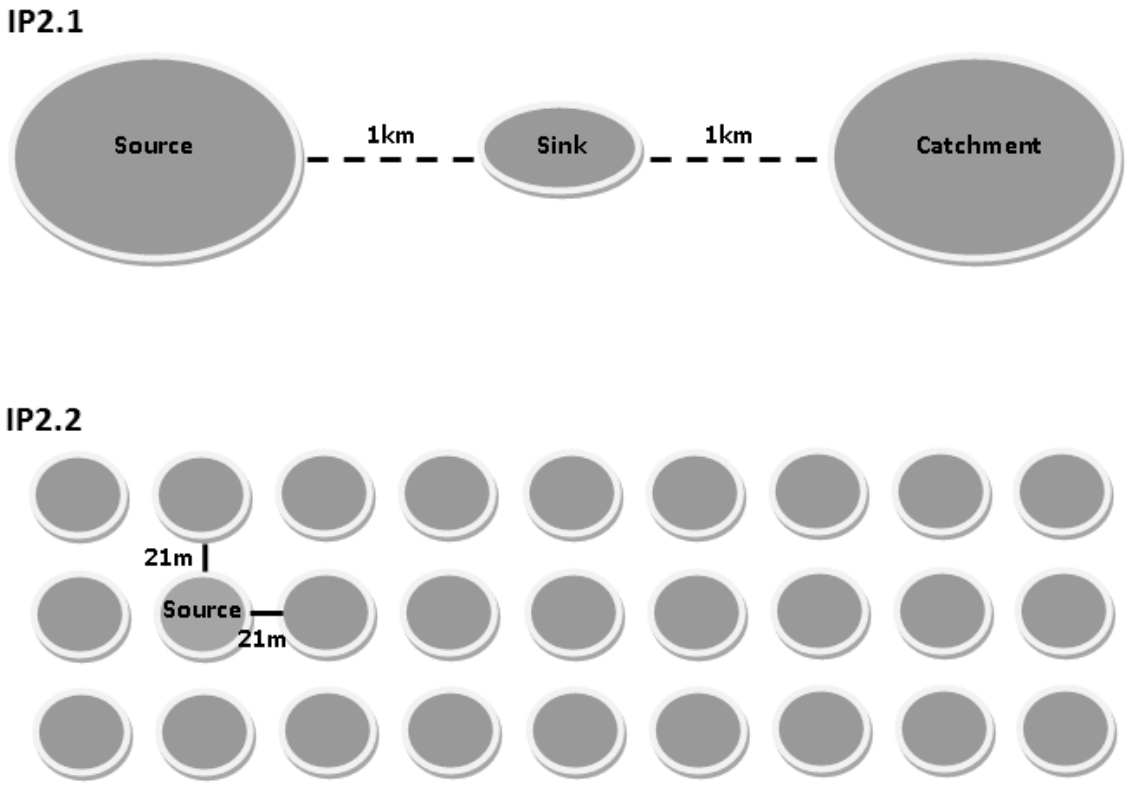

Figure 4.10. Summary of landscape configurations used for investigation (iv). Experiment IP2.1 tests the effect of increasing the size of a sink patch that is situated in between and equidistant from source and catchment population. Experiment IP2.2 incoporates a matrix configuration of 27 patches, with one nominated source patch, and patch sizes allocated random sizes between 2 and 1,000 flowers, with 100 different random combinations tested.

4.5.2.1 Experiment IP2.1

I investigated patch size effects by manipulating a single sink patch and increasing its size in subsequent trial sets. By doing this, I can compare the gene flow levels across trial sets with identical landscapes except for the size of the of the sink patch. I used a very simple three patch landscape to isolate the effect of varying the size of a sink. The other two patches are configured to be large enough to support the foraging needs of the colony if the sink patch is not deemed a suitable or sufficient foraging location. Specifically, both the source patch and the catchment patch each contain a total of 1,200 flowers. This figure is chosen based on the empirical investigation discussed in Chapter Five, after having been scaled to the bee abundance implemented here.

The third patch is placed directly between the other two patches and remains in a constant location that is 1km from each of the other patches. This third patch is nominated as the sink patch. The size of the sink patch varies according to the trial. In the first trial, the sink patch is just 100 flowers in size – only 8% of the size of the larger patches. For each subsequent trial set, the size of the sink is increased by 100 flowers up to a maximum of 1,200 flowers.

4.5.2.2 Experiment IP2.2

I explored the effect of varying patch size in a more realistic landscape configuration which explores size effects simultaneously, much as with the simultaneous exploration of distance effects in experiment IP1.3 of the first iconic pattern investigation. We know that resources are patchily distributed in real landscapes, and in this experiment I investigate a matrix representation of a foraging landscape with patches of various sizes.

The foraging landscape consists of 27 patches arranged in a 9 x 3 matrix, and every patch in the matrix is randomly assigned a patch size. Each patch is separated from its cardinal neighbours by the minimum distance of a single unit of flight time, the triviality of which means that I can isolate the effect of patch size on gene flow. All other (non-cardinal neighbour) distances are calculated geometrically, assuming that bees will take the shortest possible flight path to fly between patches.

A single patch in the matrix is designated as the source patch, and is assigned to be the patch in the third column and second row, as with experiment IP1.3 in the prior tests. This placement ensures that the source patch has nearest neighbour sinks in all directions. Each of the other 26 patches is considered as a potential sink. For this experiment, I conducted 100 separate trials and in each trial I randomly generated a set of 27 patch sizes and randomly allocated these to the matrix. Patch sizes were drawn with replacement from the range 2 to 1000 (inclusive), which ensured that I did not have single-flower patches that cannot contain both full and empty flowers, and also meant that patch sizes remained within the desired smaller patch scale.

Previously, I have used seven bees when simulating foraging landscapes of 1200 flowers and depletion and replenishment effects (bee abundance is irrelevant when depletion and replenishment are not simulated because the foragers cannot alter the experiences of other foragers). However, in the 27-patch case there are many more flowers available and by placing

too few bees in the landscape, replenishment effects will overwhelm depletion, leading to patch qualities becoming universally high and distorting the results. The experiments I outlined earlier in this chapter simulated seven bees in a landscape with 1200 flowers, based on empirical observations (Chapter Five). In contrast, this experiment contains six times as many flowers and so I multiplied the 7 bees by 6 to obtain an abundance of 42 bees for this particular experiment.

4.5.2.3 Experiment IP2.3

We see from the iconic pattern that gene flow levels decline quickly with only small increases in the number of flowers in the sink patch. I therefore conducted an experiment to attempt to manipulate this aspect of the pattern. I attempt to tune the residence for a given sink patch size by instead manipulating the rate of replenishment of empty flowers within the sink patch.

By altering the replenishment rates of the sink patch we should be able to influence the residence and thereby alter the gene flow-patch size relationship. To test this theory, I re-conducted the trials of experiment IP2.1 detailed above, but whilst I held constant the source and catchment patch replenishment intervals for empty flowers at 200 time units, I ran each trial set with sink replenishment intervals of 40, 80, 120 and 160 time units. All of the sink replenishment intervals were set to be lower than the initial value of 200 time units because, as I show in the next section, experiment IP2.1 yielded the lowest gene flow levels only at much higher patch sizes than observed in empirical investigations.

4.5.3 Results

4.5.3.1 Experiment IP2.1

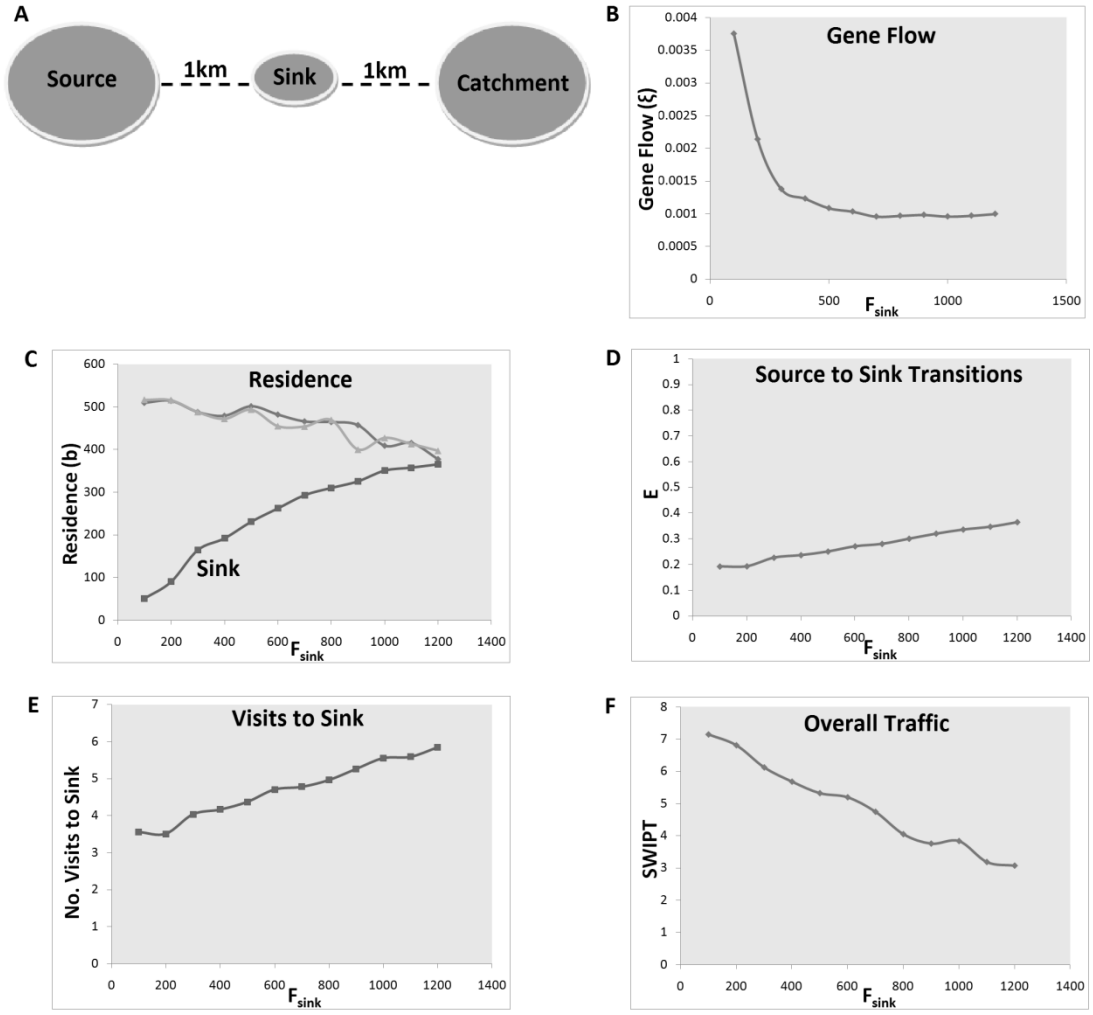

Figure 4.11. Experimental configuration and results for experiment IP2.1 testing for iconic pattern 2. A) The landscape configuration for the experiment. In each successive trial, the sink is increased in size, up to the maximum of the size of the other two patches. B) Predicted level of gene flow from the E-Psi-b model (y-axis) with increasing number of flowers in the sink (x-axis). C) Residence (number of flowers visited) of bees in each of the three patches (y-axis) with increasing number of flowers in the sink (x-axis). D) The probability of a bee visiting the sink after visiting the source (y-axis) with increasing number of flowers in the sink (x-axis). E) Number of visits to the sink patch (y-axis) with increasing number of flowers in the sink (x-axis). F) The level of System-Wide Inter-Patch Traffic (SWIPT – y-axis) with increasing displacement number of flowers in the sink (x-axis).

As the size of the sink patch increases, we find a qualitative match with the iconic pattern (figure 4.11b; figure 4.9), though the rate of decline is much slower (Klinger *et al*, 1992;

Goodell *et al*, 1997). The simulated relationship contains the effects of patch size on both *b* and *E* (figure 4.11c and d), but the effect on *E* does not generate the iconic pattern – instead, the pattern is purely driven by *b*, which overrides the influence of *E*.

4.5.3.2 Experiment IP2.2

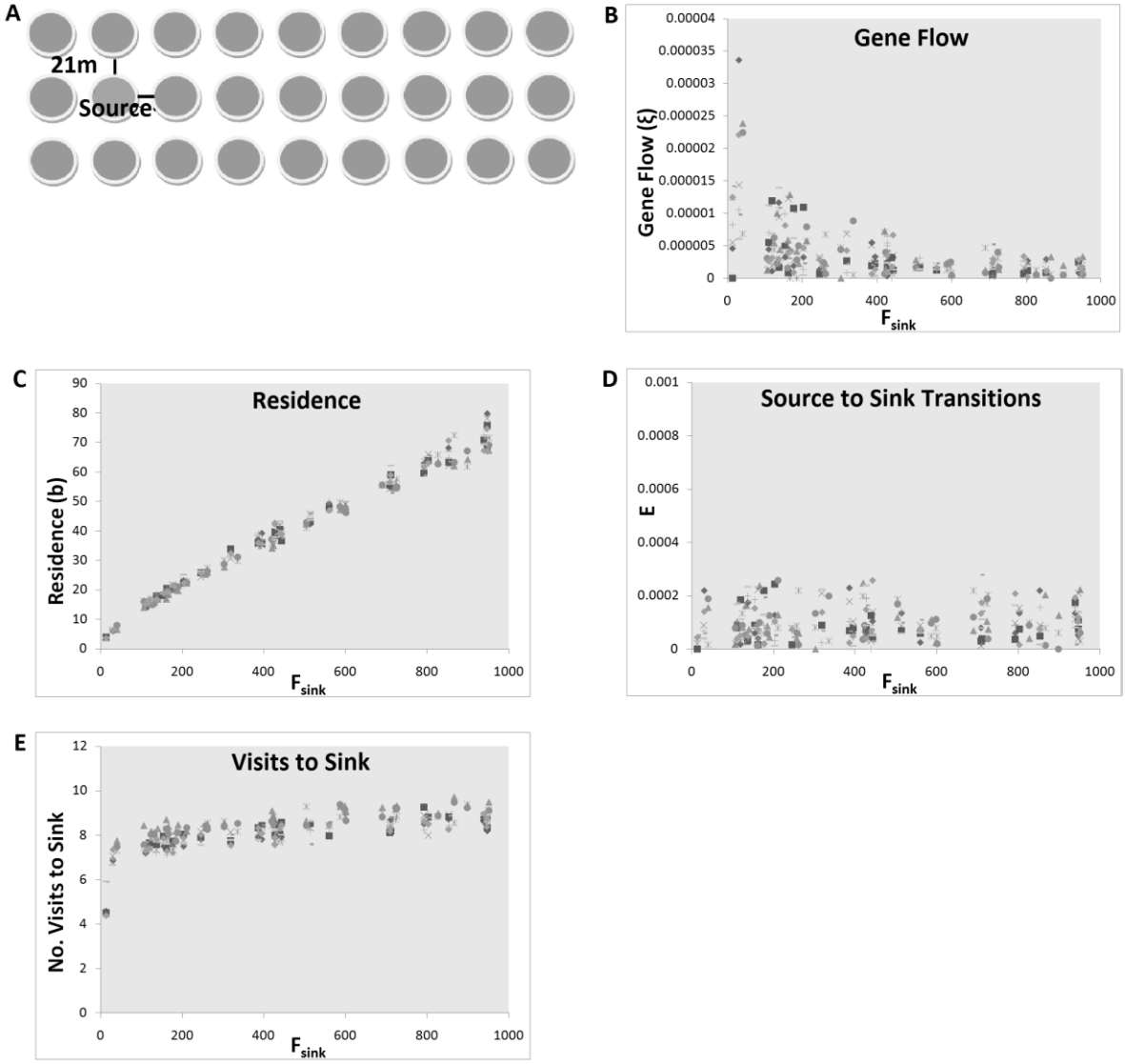

Figure 4.12. Experimental configuration and results for experiment IP2.2 testing for iconic pattern 2. A) The landscape configuration for the experiment. A single patch is nominated as the source and all other patches are in turn considered to be sinks. Patch sizes are randomly allocated across the matrix. B) Predicted level of gene flow from the E-Psi-b model (y-axis) with varying sink size (x-axis). C) Residence (number of flowers visited) of bees in each of the sink patches (y-axis) with varying sink size (x-axis). D) The probability of a bee visiting the sink after visiting the source (y-axis) with varying sink size (x-axis). E) Number of visits to the sink patch (y-axis) with varying sink size (x-axis).

The linear increase in residence with increasing size of the sink (figure 4.12c) occurs here because an increased number of flowers in a patch provides bees with more flowers; therefore a larger number of flower encounters are required to deplete a large patch's quality to the same degree as in smaller patches, and this relationship is linear. For example, in order to deplete a patch's quality by 0.1, 1 flower must be depleted in a patch of 10, 10 flowers in a patch of 100, 100 flowers in a patch of 1000 and so on.

The highest levels of gene flow were observed into the nearest neighbours of the source patch (data not shown), which we would expect as HARVEST bees prioritise moves to nearest neighbours. However since inter-patch travel is not costly in this landscape, bees do make direct moves from the source to all the sinks within the landscape and, as such, gene flow extends to some degree to all potential sinks. In comparison to the gene flow levels to nearest neighbours however, the gene flow levels to these other sinks are very low.

Again, the model is able to replicate the iconic pattern (figure 4.12b). The effect of the matrix configuration is a contributing factor that influences *E* (figure 4.12d), and so the relationship between patch size and gene flow emerges from a complex inter-play between distance effects and variation in patch sizes.

4.5.3.3 Experiment IP2.3

I found from experiments IP2.1 and IP2.2 that we could obtain qualitative agreement with the iconic pattern. However, empirical data also shows a sharp decrease in gene flow levels over a range of only very small numbers of flowers per patch.

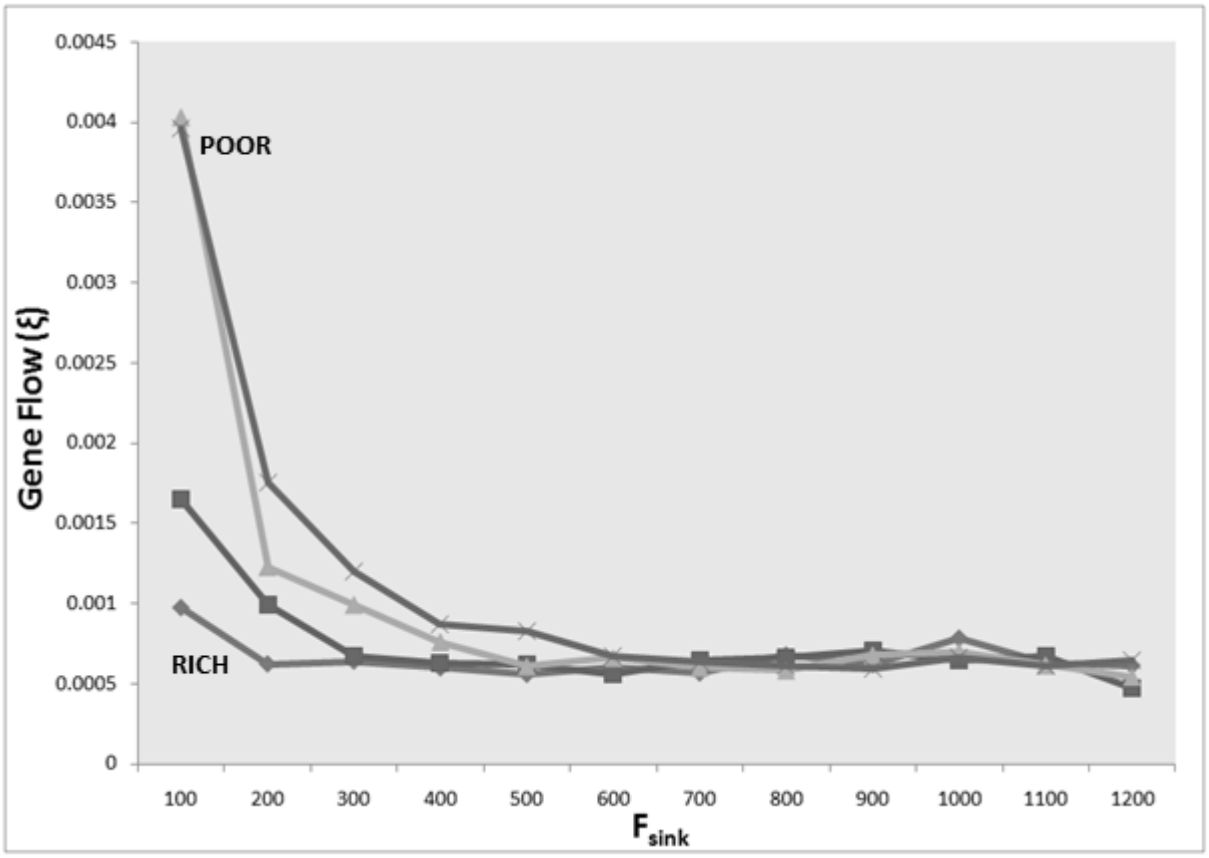

Figure 4.13. Gene flow levels from source to sink (y-axis) with increasing size of sink (x-axis) and varying rate of replenishment. Richer landscapes represent those whose sink patches replenishment more frequently. Landscape configuration as per Experiment IP2.1, with the source patch positioned directly between the source and catchment patches. Replenishment rate of sink varied, whilst source and catchment patch replenishment intervals held constant at 200 time units.

All four sink replenishment intervals tested in this experiment yield a decelerating decrease in gene flow levels with increasing size of the sink (figure 4.13), so the qualitative pattern is insensitive to the replenishment rate of the sink patch. However, gene flow reductions with patch size tend to decelerate more quickly when the rate of replenishment in the sink patch is faster. Regardless of the rate of replenishment of the sink patch, gene flow tends to the same level in large patches.

4.6 Discussion of Iconic Pattern Replication

In even the simplest realistic landscapes, HARVEST was able to qualitatively replicate the iconic pattern in which the levels of pollinator-mediated gene flow show a rapid decline with an increasing distance between source and sink populations. The simulation clarified the

mechanistic basis of this widely observed pattern by showing that the gene flow-distance relationship emerged because increased isolation of the sink patch caused increased residence (increased *b*) whilst the probability of moving to the sink patch (*E*) remained effectively constant. The simulation also predicted previously unforeseen context-dependent effects of landscape configuration on gene flow, but only in landscape configurations that are unrealistic or, at least, rare (i.e. when a sink patch is increasingly separated from the source patch without approaching an additional third patch of flowers). Specifically, increased gene flow levels arose when the sink field was furthest away from the source field if the sink field was close to another field. In realistically complex landscapes with a matrix of patches, however, the iconic pattern re-emerges because of the presence of additional patches.

Overall, two key points can be deduced from the results of these experiments. First, the model is capable of reproducing the iconic pattern of increased distance resulting in a decelerating decrease in gene flow levels when it is implemented in realistic experimental configurations used in empirical studies that have investigated this phenomenon. This provides the model with increased credibility as a predictor of bumble bee patch choice behaviours and resultant bee-mediated gene flow levels. Second, the experiments here have shown that the iconic pattern as it has been previously expressed in general terms - of increasing distance between source and sink equating to decreased gene flow levels - is insufficient to describe gene flow in certain landscapes. In very simple landscapes with only a few, isolated patches, I found that by making only a relatively minor change to the landscape configuration, the iconic pattern of gene flow disappeared. This finding advocates the use of modelling to test the generality of proposed theories from comparatively narrow empirical data sets.

Furthermore, the model qualitatively reproduced a second iconic pattern - a decelerating decrease in gene flow levels with increasing size of the sink patch. Gene flow was less sensitive to change in the patch size until much higher patch sizes than those observed empirically, however, but only in resource-rich landscapes.

Overall, out of the first set of experiments, experiment IP1.3 would seem to be the most realistic approximation of many real world foraging landscapes (Cresswell, 2006), as it considers the influence of complex matrices of background patches. Experiments IP1.1 and IP1.3 qualitatively replicated the first iconic pattern, implying that the model is capturing aspects of optimal foraging employed by real bumble bees, that are essential for the gene flow-distance pattern to emerge.

Experiment IP1.2 did not replicate the iconic pattern. Whether or not this experimental configuration is representative of any real world foraging scenarios is debatable, but nonetheless the result brings into question the universality of the iconic gene flow-distance pattern, and I would recommend further investigation into this area.

The model successfully replicated the iconic gene flow-patch size relationship in both experiments. The linear relationship between residence and patch size demonstrated by experiment IP2.2 does not match observations in the empirical studies that instead reveal a decelerating increase in residence with increasing patch size (Klinger *et al*, 1992; Goodell *et al*, 1997). However, it is possible that the nature of the empirical study configurations actually more closely approximate the configuration used in experiment IP2.1, which did yield a realistic residence-patch size relationship. Indeed, the extremely small scales of the patches used in the empirical investigations may suggest that bees considered these to be smaller scale patches sitting amongst much larger patches or even small fields nearby in the landscape, which is the scenario captured by experiment IP2.1.

The bees simulated here did not employ systematic foraging mechanisms. By implementing such a mechanism in the model it is possible that the residence results of this experiment would replicate the realistic residence-patch size relationship, because the avoidance of a specified number of recently visited flowers (essentially a constant-sized memory span) should decrease the frequency of empty flower encounters and therefore lengthen residence. This effect would be less significant in larger patches than smaller patches, and therefore saturation in residence increases could be observed. I discuss the potential extension of the model to accommodate systematic foraging in more detail in Chapter Seven.

The results of experiment IP2.3 demonstrate that the patch size-gene flow relationship can be tuned by modifying the rate of replenishment of the sink patch. This could enable accurate quantitative agreement between the model behaviour and empirical results if measurements of actual levels of nectar and pollen production were incorporated. Where replenishment rates have not been measured in empirical investigation, there is a strong argument for tuning the model in this way to obtain quantitative agreement.

4.7 Summary

Overall, I have shown in this chapter that the model is able to successfully replicate two iconic patterns relating to bumble bee mediated gene flow levels and across a variety of landscape configurations. Since these are commonly observed patterns in natural systems, I find reassurance that the model could be a powerful predictor of bumble bee foraging patterns. In the next chapter, I look at a more direct test of the patch choice algorithm, which compares the model predictions against an empirical experiment that I designed and conducted specifically to assess the power of the model to predict residence and traffic-based bumble bee behaviours. This is an attempt to further validate the model as a useful simulation.

Chapter Five : An Empirical Test of HARVEST

5.1 Introduction

For maximum credibility, the development of any ecological model should incorporate empirical validation to assess the model's capabilities at predicting the real world behaviours being simulated. For example, the Threshold Departure Rule (Hodges, 1985a) was validated by the author, who compared the predictions generated by the rule with observed movement behaviours of queen bumble bees (Hodges, 1985a). The Marginal Value Theorem (MVT) (Charnov, 1976) has been extensively tested because of its widespread use across ecological domains; some tests have found MVT to be a useful predictor of real world foraging behaviour (Cibula and Zimmerman, 1984; Pleasants, 1989), whilst others have found only conditional agreement (Pyke, 1978) or poor quantitative accuracy (Giraldeau and Kramer, 1982).

Some authors present the validation of their models in terms of the discrepancy between the predictions of the model and a theoretical optimum foraging scenario (Bernstein *et al*, 1988; Groβ *et al*, 2008). This validation is based on the premise that real world foragers are optimal, and therefore the model can be validated by its ability to replicate a perfect forager. This may be problematic, however, as real-world foragers are unlikely to be perfectly optimal (Pierce and Ollason, 1987). Whilst such comparisons with theoretically optimum foraging are useful for a first approximation of the potential of the model (as I demonstrated in Chapter Three), it is useful to validate models against empirical data.

In this chapter, I present the details of my own empirical study, which was designed to be a direct test of the patch choice behaviour predicted in the model. My goal is to generate predictions that can be used to solve the *E*-Psi-*b* gene flow model (Cresswell *et al*, 2002) when applied to the landscape-scale foraging of bumble bees. We therefore need confidence that the residence and inter-patch traffic predictions that the model generates are accurate. I designed a simple patch-scale test landscape that could be replicated both empirically and *in silico* so that I could directly test the model's predictions. Whilst we cannot directly validate the efficacy of the model's landscape-scale predictions, it would seem reasonable to assume that the fundamental principles and economic constraints of bumble bee foraging behaviour may translate as spatial scales are increased (Burns and Thomson, 2006). Certainly there seems to be no additional factor that will offset the costs of travel, which means that HARVEST is unlikely to underestimate the extent of foraging movements.

5.2 Methods

5.2.1 The foraging landscape

I constructed a landscape of six foraging patches on a grassy field within the grounds of the University of Exeter in June 2008. The site was chosen because of its propinquity to wild flowers and bushes that attracted bumble bee activity, though analysis of bee behaviour was restricted to the six patch landscape. Each patch consisted of approximately 20 *Brassica napus* plants contained in two large moveable plastic trays placed side by side. Patches were laid out in a 2 x 3 grid matrix, with patches aligned in their respective rows and columns. Patches were separated from their nearest neighbours by a distance of 6 metres.

All patches on the field – which I designated 'W' (Wet) patches - were watered daily for the duration of the experiment to maintain their health. Nectar production rate is known to respond to soil moisture levels (Wyatt *et al*, 1992; Carroll *et al*, 2001), and in order to introduce variation in patch quality, I maintained a further three patches - six trays of plants – that were designated 'D' (Dry) patches and kept on a dry bench of a nearby glasshouse. These patches were deprived of water from three days before the start of the experiment, which brought the plants to their wilting point, and were not watered for the duration of the experiment in an attempt to create inferior quality patches.

I tested two different types of landscape within the experiment – 'homogenous' and 'variable'. In a homogenous landscape all six patches in the matrix were W patches, whilst a variable landscape consisted of three W patches and three D patches. To create a variable landscape, I randomly selected three W patches to be swapped out of the matrix and replaced them with the three D patches. After any session in which a variable landscape was tested, the patches were swapped back, unless the next session also required a variable landscape configuration. D patches were removed from the glasshouse and placed in the matrix for at least one hour before observations were made with them, to ensure that they were exposed to ambient levels of bee activity comparable to the W patches.

5.2.2 Data capture

The experiment was divided into eight sessions taking place over five days from 2nd June 2008 to 6th June 2008 inclusive. Typically, each day contained two sessions, one taking place in the

morning and one in the afternoon, but heavy rain showers on the afternoon of 2nd June meant that only one session was conducted that day, so an extra session was conducted on 6thJune. Wet weather conditions are not suitable for bumble bee observations as they do not forage during such conditions (Peat and Goulson, 2005; Cartar, 1992). On all other days, the weather was generally sunny and warm, and always dry. Each session was 2 hours long, with morning sessions commencing at 10.00 and afternoon sessions commencing at 14.00. During each session, either one or two observers were in the field simultaneously. Each observer was provided with a randomised sequence that determined the order in which they should observe each of the six patches.

At the start of a session, each observer would move to stand approximately 2m away from the first patch in the visit sequence, and outside of the grid itself so as not to influence bee movements. The observer recorded the arrivals, duration of stay, and departures of bumble bees within the chosen patch for a total of 10 minutes. Observation data was dictated into voice recorders and recorded on log sheets. In some cases, one or more bees were already present in the patch at the start of the observation period, or a bee remained in the patch at the end of the observation period. Such instances were recorded and the arrival and departure times were assumed to be 00:00 and 10:00 respectively. At the end of the 10-minute observation period, the observer moved onto the next patch in the randomly generated sequence and repeated the process for this patch. This continued until all six patches had been visited once by the observer, at which point the observer repeated the same sequence of visits once more, so that all patches were observed twice by each active observer within any given session.

Simply by recording the arrival and departure times within each 10 minute observation, we can determine arrival numbers and patch residence, and estimate overall levels of bee effort per patch. I calculated the residence of a bee in a patch by first calculating the difference between the arrival and departure times of the bee within the patch to obtain a time-based residence. I then converted this residence time into a predicted number of flower visits, as both the model and the E-Psi-b gene flow model require residence to be expressed in flower visits. To apply the conversion, I took sample readings of flower handling time - the time taken for bumble bees to sample a single flower - by timing the duration of approximately 20 sequential visits. After the experiment, the time difference between the first and final flower visits of a sample was divided by the number of flowers visited in the sample to calculate a mean flower handling time. Overall mean flower handling times were used to convert time-based residence data to flower-based residence data. I calculated overall bee foraging effort for each patch by multiplying average patch residence by the number of arrivals into a patch. To accommodate

the difference between patch sizes in variable landscapes, I express this overall bee effort for each patch as the average bee effort per flower both for the empirical and model data.

Out of the eight 2-hour sessions in the experiment, four were conducted with homogenous landscapes and four with variable landscapes, but the observation procedure was identical in both. To characterise patterns of inter-patch movement in the grid, an additional observation session was carried out. A single observer randomly selected a patch and followed the first bee that arrived at that patch. The observer recorded the sequence of patches visited by the tracked bee, along with the number of flowers sampled in each patch. This continued until the bee left the experimental grid, at which point the process was repeated.

5.2.3 Quantifications of key parameters

DAY 1		**S.E.**
Mean Flowers Per W Patch:	115	*9.2*
Mean Flowers Per D Patch:	154	*22.7*
DAY 2		
Mean Flowers Per W Patch:	105	*3.4*
Mean Flowers Per D Patch:	138	*4.7*
DAY 3		
Mean Flowers Per W Patch:	116	*4.9*
Mean Flowers Per D Patch:	139	*17.7*
DAY 4		
Mean Flowers Per W Patch:	129	*2.7*
Mean Flowers Per D Patch:	154	*8.5*
DAY 5		
Mean Flowers Per W Patch:	136	*4.7*
Mean Flowers Per D Patch:	Not Taken	*N/A*
ACROSS EXPERIMENT		
Mean Flowers Per W Patch:	120	*3*
Mean Flowers Per D Patch:	147	*6.9*

Table 5.1. Average number of open flowers per patch on each day of the empirical study, along with the type of patch (W or D) to which the flower count relates. No D patch flower counts were taken on day 5 as no further variable landscape sessions took place after day 4. Mean W and D patch flower counts across the study are also shown.

After the morning session of each of the first four days of the experiment, I counted the number of open flowers in each of the six W and three D patches. On average, the D patches contained more open flowers than the W patches, with a mean of 147 flowers per D patch compared to 120 per W patch (Table 5.1), probably because the additional time the D patches spent in the

better environment of the glasshouse caused additional growth in these patches, and although they were deprived of additional water for 3 days before the start of the experiment, it would have taken some time for the plants to wilt (Kramer, 1950). However, variation in the numbers of flowers in patches is not problematic for HARVEST, because variable flower numbers can be specified in the model.

5.2.4 Nectar production

The goal of the drought manipulation was to affect the nectar yields rather than the absolute number of flowers available. Drought affects nectar production in plants (Wyatt *et al*, 1992; Carroll *et al*, 2001), and commonly causes wilting of the leaves, which is an indicator of drought stress (Kramer, 1950; Mogensen *et al*, 1996). A separate study showed that nectar production rates halved under the drought conditions that I applied (figure 5.1).

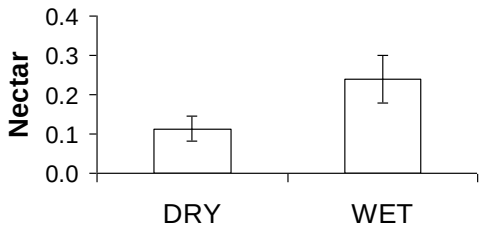

Figure 5.1. Mean nectar production rates (microlitres h^{-1} – y-axis) in flowers of droughted ('D') and watered ('W') individuals (x-axis) of B. napus *under glasshouse conditions. Droughted plants produced nectar significantly more slowly (one-tailed t-test, t = 1.76, df = 14, P<0.05) (Cresswell, unpublished)*

5.2.5 Bee foraging rate

Three species of bumble bee were observed foraging during the experiment. By far the most common visitor was *Bombus lapidarius* (henceforth BL), constituting 79% of all bumble bee visits during the experiment. I found that the mean flower handling time for BL on *B. napus* was around 2.1 seconds. The second most common visitor was *Bombus terrestris* (BT) which constituted 16% of all bumble bee visits and which I found to be the slowest forager, with a mean flower handling time of 2.5 seconds. The least frequent visitor was *Bombus pratorum* (BP) which made up just 5% of bumble bee visits and was the fastest forager with a mean flower handling time of 1.8 seconds.

5.2.6 Parameterization of the model

Only critical descriptive variables of the experiment were supplied to the model, such as the number of patches, their arrangement, the number of bees present and so forth. I did not provide the model with any of the response, or output, variables from the empirical study, such as residence or inter-patch traffic results. Therefore, if the model replicates my empirical findings, the model is validated. I set the parameter values of the model to reflect the configuration of the empirical setup as follows (also see Table 5.2).

As with the empirical study, a landscape of six patches was configured in the model, arranged in a 2 x 3 matrix with two rows and three columns. In the empirical study, the distance between cardinal neighbour patches was 6m. In the model we express inter-patch distance as the time required for a bee to travel between the pair of patches. Previous studies have shown bee flight times between close but spatially distinct patches to be around 2.5 seconds (flight duration in seconds = 0.2592(distance in m) + 0.7475; Cresswell, unpublished data). Inter-patch flight times were therefore taken to be 2.5 seconds, with all other inter-patch distances calculated geometrically in proportion. I specified the time to handle a single flower as 2 seconds, which approximated the observed mean time to handle a single flower of 2.1 seconds. When comparing model residence data with empirical data, the time-based empirical data was divided by 2 to estimate the number of flowers visited. To increase the resolution at which I could parameterise the model here, a 2 second flower handling time was represented as 20 time units, thereby changing the definition of a single (and atomic) time unit in the model to be 0.1 seconds. This tenth-of-a-second resolution allowed me to accurately configure a 2.5 seconds inter-patch distance as 25 time units. To put it another way, by moving to a nearest neighbour patch we see that a bee sacrifices the opportunity to sample a single flower and incurs an additional penalty of 0.5 seconds.

Patch sizes could be precisely replicated by the model as I had taken accurate flower counts for each patch in the empirical study. The mean number of flowers in W patches across the experiment was 120.3, and so each W patch in the model contained 120 flowers. The mean number of flowers in D patches was 146.6, so I used a total of 147 flowers for each D patch. Homogenous and variable landscapes were configured as per the empirical trial. For initial conditions, I assumed each patch in the landscape presented 50% quality, so that half of each patch's flowers were full and half were empty.

I simulated depletion and replenishment effects because at the scale of small patches, the foraging impact of bees on standing crop levels is likely to be significant. As the model's reward system is binary and I have no direct measurements of replenishment frequency within the six-patch empirical study, I derived the replenishment frequencies as follows. Real bees often exhibit traplining behaviour in which sequences of visits to flowers and patches are repeated (Thomson, 1996; Ohashi *et al*, 2008; Williams and Thomson, 1998), which is a form of systematic foraging. For the purposes of determining replenishment frequency here, I assume that a flower will have fully replenished at least by the time a bee could have performed a perfect systematic foraging bout within the grid, visiting all flowers in the grid once, because this is the minimum replenishment rate required to sustainably support an optimally harvesting bee in a closed system (separate pilot studies showed that this replenishment rate sustainably supprted two RL bees without long term diminution of system-wide reward levels – data not shown). Formally :

$$I = (F \div B) \times h \qquad \text{(Eq 5.1)}$$

where I is the replenishment interval to be calculated, F is the total number of flowers in the landscape, B is the number of bees in the simulation, and h is the flower handling time (in time units).

Equation 5.1 provides an infrequently replenishing scenario, because the flowers are replenished at a rate equivalent to a single bee visit on a perfect trapline. Therefore I used I to define the replenishment interval of the infrequently replenishing D patches. As I discussed above, W patches generate nectar twice as fast as the D patches (figure 5.1). The replenishment interval for W patches was therefore taken as half the D replenishment interval.

In a variable landscape with both W and D patches, there are three W patches each containing on average 120 flowers and three D patches each containing on average 147 flowers. So the total number of flowers in the landscape (F) is 801. All foraging activity in the model is constrained to the simulated matrix of patches. To determine the number of bees to simulate in a closed model system, I estimated the number of bees that were foraging simultaneously within the matrix in the field study. I extracted from the collected data the proportion of observation

time that was spent actively observing foraging bees. I then scaled this up according to what proportion of the matrix was being observed simultaneously, which itself was dependent on the number of observers in a session. Formally :

$$B_E = (T_B \div T) \times (1 \div a) \qquad \text{(Eq 5.2)}$$

where B_E is the estimated number of bees foraging simultaneously in the matrix, T_B is the total time (in seconds) spent actively observing bee activity in a session, T is the total time spent observing the matrix (regardless of whether a bee was present or not) and a is the proportion of the array being observed during a given session. a is 1 / 6 for sessions with a single observer and 1 / 3 for sessions with two observers.

Equation 5.2 was used to estimate the number of bees in each of the eight sessions of the empirical study to obtain an estimated number of bees foraging simultaneously on the matrix in each session. I then calculated the average across the empirical study and found a mean of 1.5 bees foraging simultaneously (with a standard error of +/- 0.2). Since this figure lies between one and two, and since we should only simulate integer colony sizes in an individual-based model framework (Grimm and Railsback, 2005), I initially simulated two bees, but made further investigation with just one bee foraging simultaneously in the model.

Solving equation 5.1 with F = 801, B = 2 and h = 20 (20 time units = 2 seconds in this experiment), I calculated a replenishment interval for D patches of 8010 time units. For simplification I rounded this down to 8000 time units. Therefore, a W patch replenished in 4000 time units.

In line with the experiments detailed in the chapter, I set the fixed nectar carrying capacity of model bees to be 500 nectar units, and provided them with a β value of 0.05. I again imposed that the nest should be equidistant from all patches in the matrix to negate potential bias from explicit nest placement when the true location of the nest is unknown.

Parameter	Identifier	Value
L	Number of patches in landscape	6
t(i, j)	Flight time between patches i and j	2.5 seconds between neighbours
Q	Initial quality of patches	50%
F_W	Number of flowers in W patches	120
F_D	Number of flowers in D patches	147
I_W	Replenishment interval for W patches	400 seconds
I_D	Replenishment interval for D patches	800 seconds
B	Number of bees in the grid	2
C	Bee's nectar carrying capacity	500 full flowers
h	Flower handling time	2 seconds
β	Sensitivity to individual flower sample	0.05

Table 5.2. Summary of parameterisation for the model's replication of the empirical experiment.

5.2.7 Implementation of HARVEST

I ran two trial sets – one for each landscape type (homogenous and variable) – with each trial set comprised of ten separate trials. I allowed bees to forage for three hours in each trial (108,000 time units, or handling times), which was equivalent to the two hours observation time and one hour equilibration time for each session in the empirical study.

To replicate the additional session at the end of the empirical study that analysed inter-patch movements, consecutive patch visit data from the level of inter-patch traffic per foraging bout in the model was examined. Since the model bees operate in a closed system and cannot leave, consecutive patch visits were analysed over a time horizon equal to the length of a foraging bout. This is the closest approximation in the model to a single bout in the field array, and

whilst it is entirely possible that the real bees were not returning directly to the nest after having left the field array, I believe this comparison to be sufficient for first approximation. I extracted this data from the homogenous landscape scenario trial set, as the session in the empirical survey was conducted with a homogenous landscape of six W patches.

5.3 Results

5.3.1 Empirical study results

Bees spent quite a long time foraging among the patches of my array (figure 5.2; mean = 9.4 patches visited, S.E. = 1.5), which suggests that they would have had an opportunity to learn and select among the patches based on their quality.

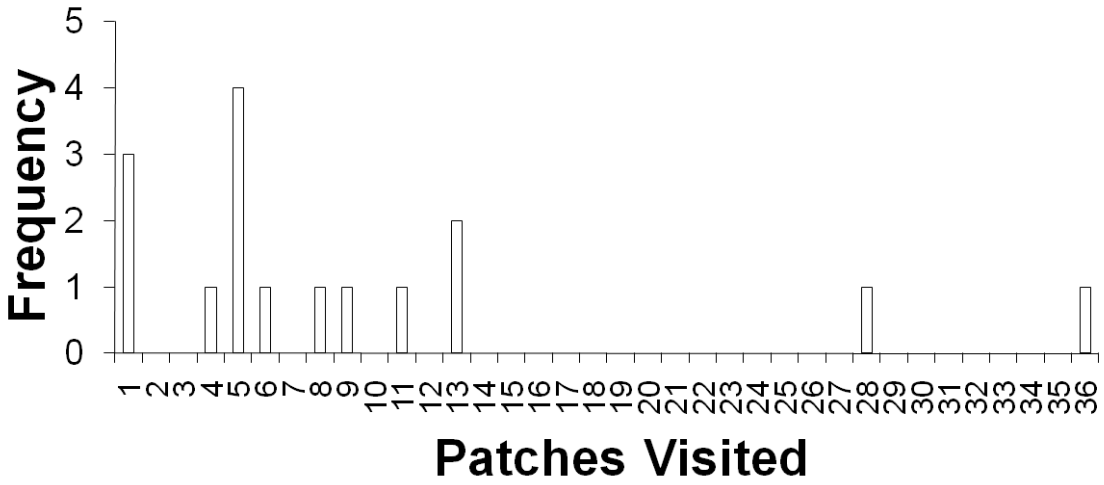

Figure 5.2. The frequency of total successive patch visits during a single foraging bout on the array in the sample observation session. 16 bees were followed individually as they entered the array, and the number of patches that they visited before leaving the array were recorded (x-axis). Y-axis shows the number of bees that made each specified number of patch visits (y-axis).

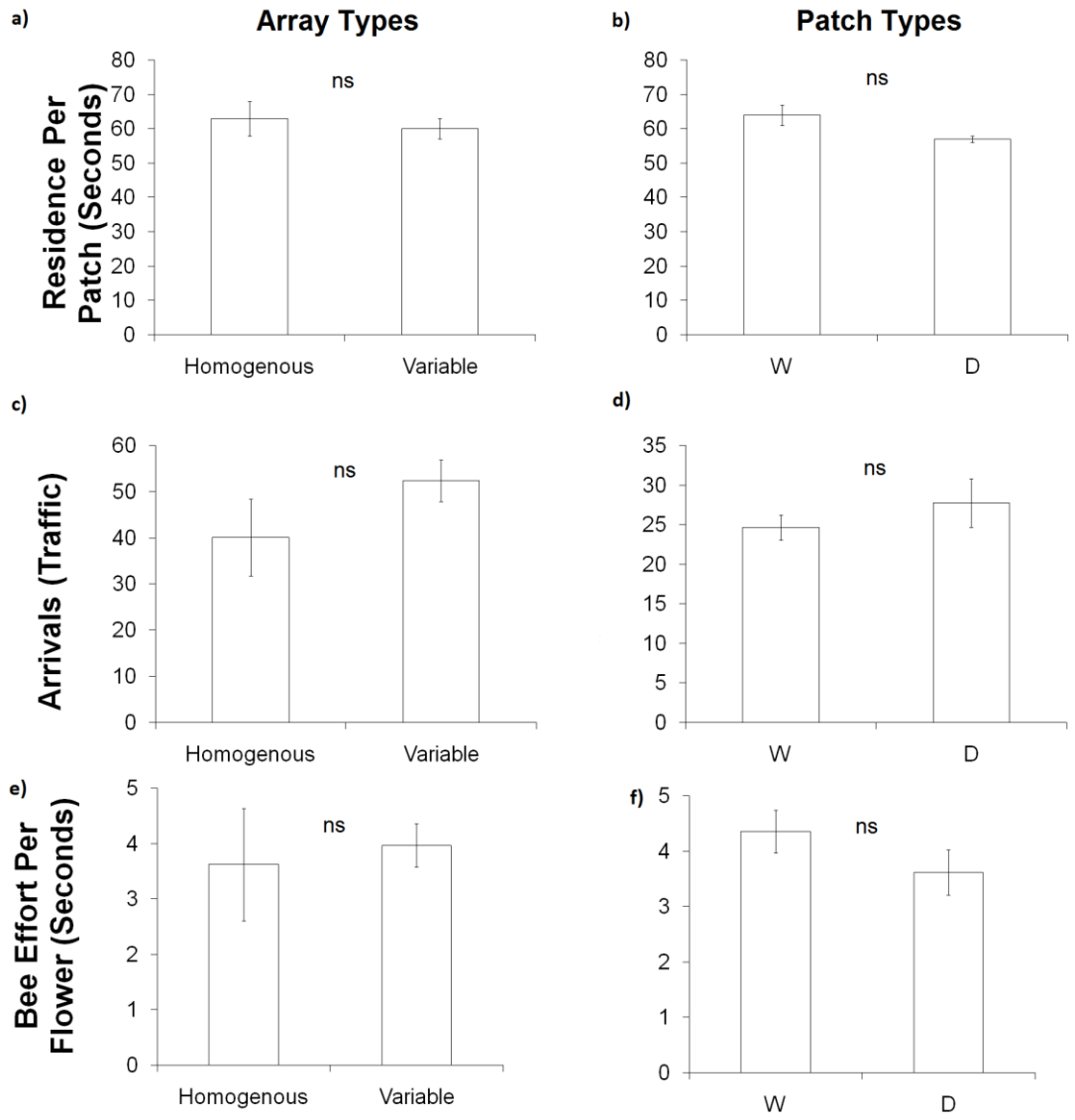

Figure 5.3. Mean patch residence (in seconds), arrival rates, and overall bee effort (y-axis measurements) of real bees in the empirical array of six patches, comparing homogenous and variable landscapes, and W and D patches (x-axis measurements). Averages taken across all patches of each type, and across all sessions. Error bars indicate ± 1 standard error among sessions.

Mean residence did not differ significantly between homogenous and variable landscapes (independent samples t-test, df = 3, t = 0.52, P = 0.64; figure 5.3a). The similarity between landscape types in mean residence probably arises because the residence-decreasing effect of poor quality D patches in variable landscapes is offset by the residence-increasing effect of W patches receiving more foraging attention (figure 5.3f).

In variable landscape sessions, the mean residence did not differ significantly between W and D patches (independent samples t-test, df = 4, t = 1.49, P = 0.21; figure 5.3b). On average, there was more inter-patch traffic in variable landscapes than homogenous landscapes, but again this difference was not significant (independent samples t-test, df = 5, t = -1.29, P = 0.25; figure 5.3c). Variable landscapes contain three poorer quality patches, where bees would less likely settle. Bees tended to visit all patches in variable landscapes with similar frequency (independent samples t-test, df = 5, t = -0.91, P = 0.41; figure 5.3d). Bee effort per flower was very similar in both variable and homogenous landscapes (independent samples t-test, df = 4, t = -0.32, P = 0.77; figure 5.3e). Bee effort in W patches was slightly greater than in D patches, but again did not represent a significant difference (independent samples t-test, df = 6, t = 1.33, P = 0.23; figure 5.3f).

In the final observation session in which inter-patch movement patterns were analysed, a total of sixteen bees were tracked. These sixteen bees collectively visited 151 patches and sampled 4,996 flowers. On average, bees visited around nine to ten patches in sequence before leaving the grid, with a mean of 9.4 patches and a standard error of 2.4. Additionally, the mean number of flowers probed by a single bee was 312, though with a large standard error of 85.5 indicating a significant amount of variation in these results. Notably, two thirds of all patch visits and almost three quarters of all flower visits were the contribution of just five individual bees. These five bees visited on average 20.2 patches in sequence, with the longest sequence being 36 patches in length, and they visited a mean of 737 flowers with the highest number of flowers visited being 1,204.

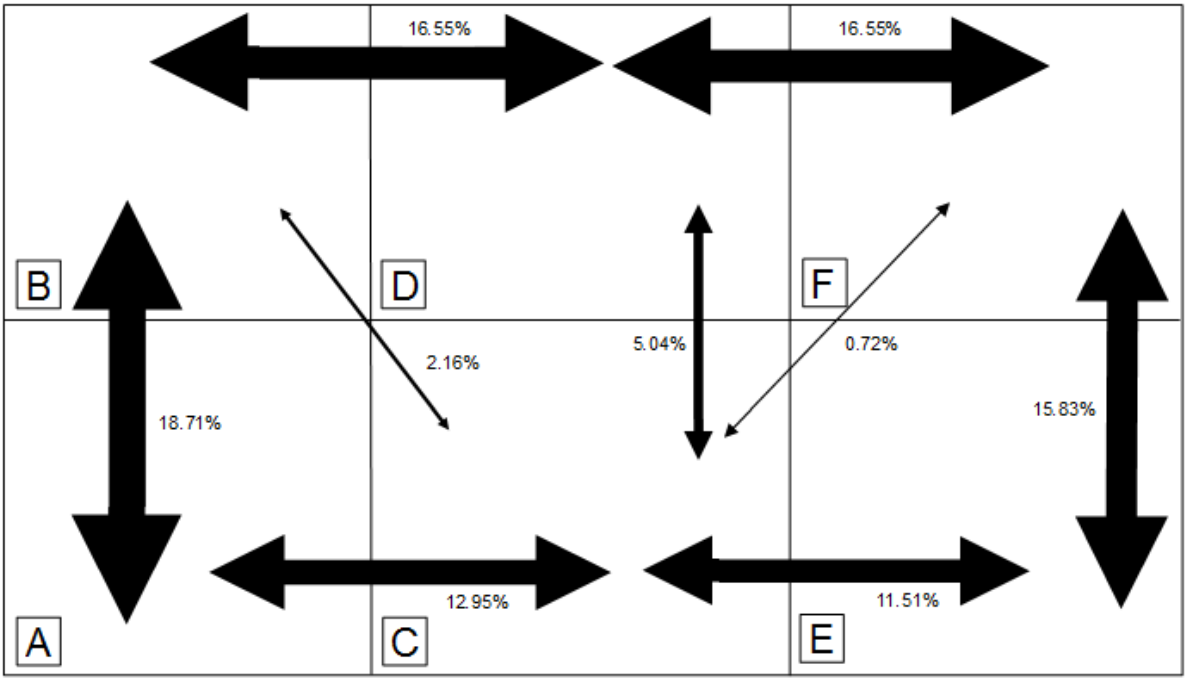

Figure 5.4. Frequency of specific inter-patch movements made by real bees in the final observation session of the empirical study, in which 16 bees were followed and their movements within the array recorded. Arrows indicate the transitions between pairs of patches. The thickness of the arrows indicates the frequency of the inter-patch transition, with thicker arrows representing higher frequencies. Percentage annotations indicate the percentages of specific transitions made by the sixteen made, with percentages rounded to 2 d.p.

Nearly all inter-patch movements were made to nearest neighbour patches (97.12%). All other moves were diagonal moves between patches. No moves were observed to beyond one patch in neighbourhood depth (figure 5.4). It seems that bees favour shorter moves, which supports previous findings (Zimmerman, 1979; Zimmerman, 1981; Gegear and Thomson, 2004) and the notion that bumble bees are economically motivated (Heinrich, 2004).

5.3.2 Model results

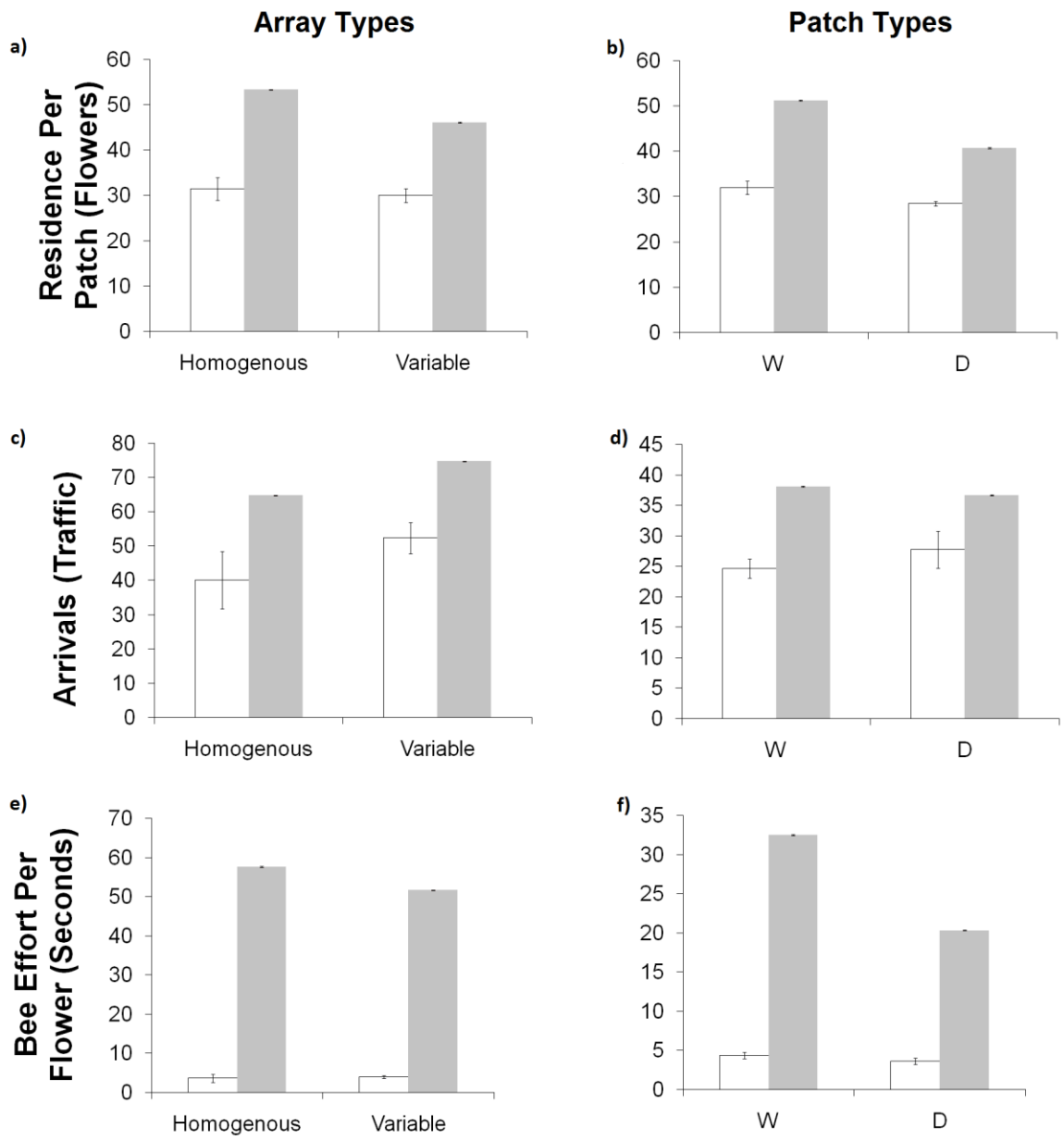

Figure 5.5. Mean patch residence (in flower visits), arrival rates, and overall bee effort (y-axis measurements) of bees, comparing real bees (empirical data – open bars) and HARVEST bees (model predictions – closed bars). Comparisons shown between homogenous and variable landscape configurations, and W and D patches. Empirical residence data converted to flowers visited by assuming one flower visit for every 2 seconds of residence. For empirical data, averages taken across all patches of each type, and across all sessions. For model data, averages taken across all patches of each type, and all 10 trial replicates. Error bars indicate ± 1 standard error among sessions or trials.

HARVEST failed to quantitatively predict the residence and arrival rates of real bees, having overestimated the observed data (figure 5.5a-d). Bee effort prediction suffers an even larger discrepancy (figure 5.5e-f), as it is the product of two overestimates. HARVEST did replicate some qualitative patterns between landscape scenarios, but none of these patterns were found to be significant empirically, and so this finding is unlikely to be significant.

Model bees did not trapline or move between patches with any regular pattern (figure 5.7), and often bypassed nearest neighbours when D patches were introduced in variable landscapes (figure 5.8).

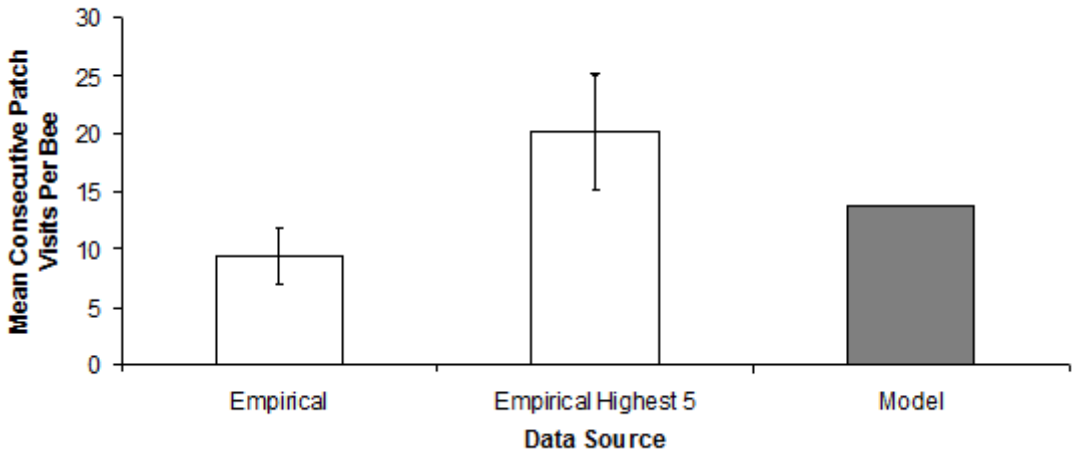

Figure 5.6. Mean number of consecutive patches visited by bees upon visiting the array (y-axis). Counts taken from 16 bees until they departed from the array. X-axis categorises the data-sets : 'Empirical' is the full set of 16 bees from the empirical session, 'Empirical Highest 5' is the set of data taken only from the 5 real bees that visited the most number of consecutive patches in the empirical session, and 'Model' is the full set of two bees that were simulated in HARVEST. Error bars indicate +/- 1 standard error among observations.

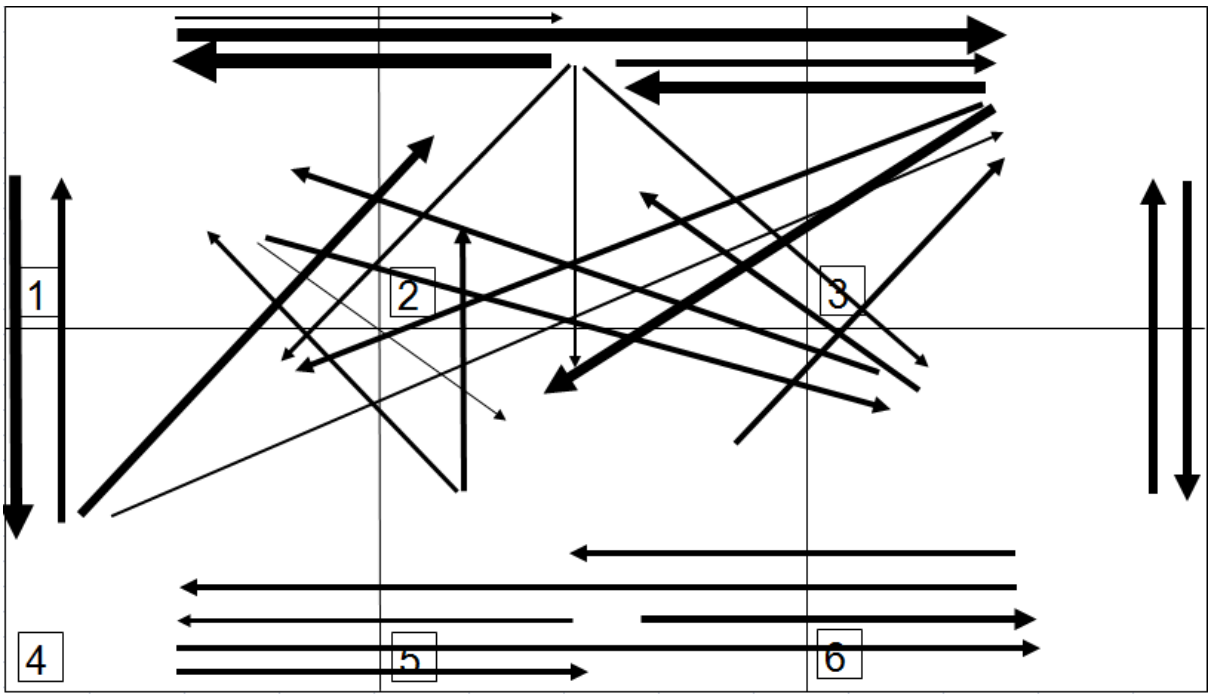

Figure 5.7. Frequency of specific inter-patch movements made by HARVEST bees in a sample homogenous trial, in which the inter-patch movements of 2 bees were extracted from the simulation's results. Arrows indicate the transitions between pairs of patches. The thickness of the arrows indicates the frequency of the inter-patch transition, with thicker arrows representing higher frequencies.

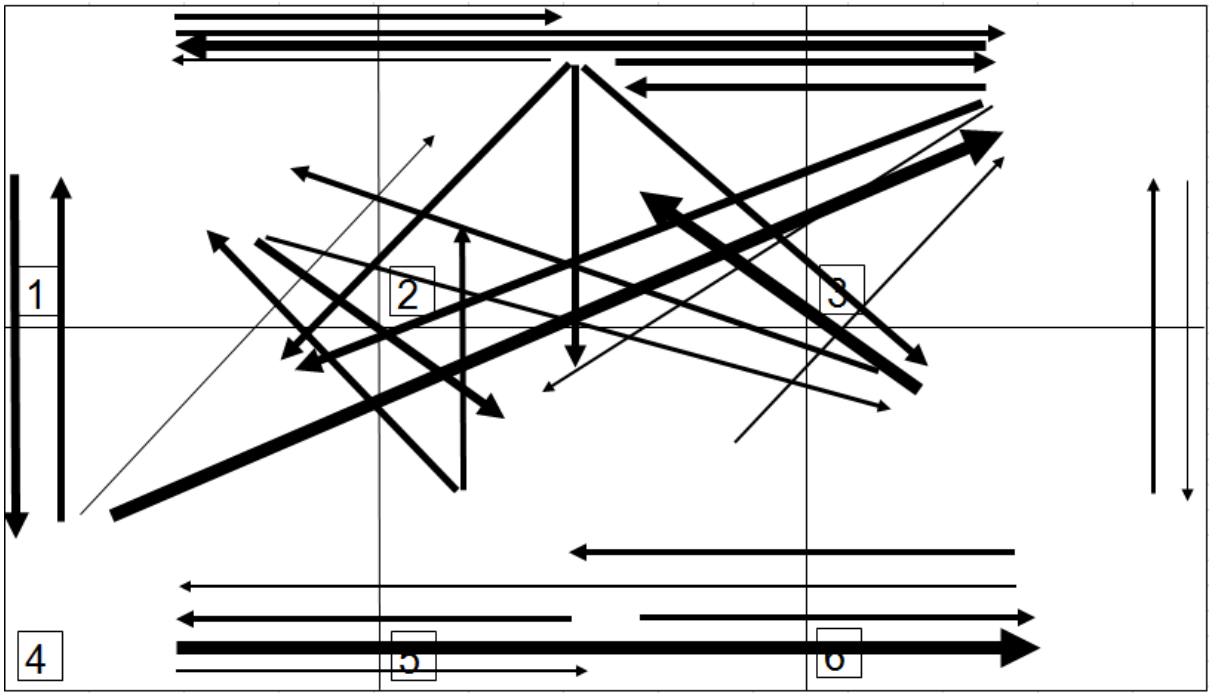

Figure 5.8. Frequency of specific inter-patch movements made by HARVEST bees in a sample variable trial, in which the inter-patch movements of 2 bees were extracted from the simulation's result. Arrows indicate the transitions between pairs of patches. The thickness of the arrows indicates the frequency of the inter-patch transition, with thicker arrows representing higher frequencies. D patches are patches '2', '4' and '6'.

5.4 Sensitivity analysis of results to variation in the number of simulated bees

5.4.1 Sensitivity analysis of results to variation in bee abundance

To test if the size of the simulated colony had affected the residence and arrival results, I ran the HARVEST experiments twice more – once with the number of bees simply reduced to 1, and once with the number of bees reduced to 1 and the replenishment interval doubled in accordance with equation 5.1 – to see if an over-estimate in the number of bees led to an over-estimate in residence and arrival rates. In fact, I found that by simulating just a single bee, the over-estimates were worsened in comparison to the empirical data (figure 5.9). This implies that the over-estimates were not due to over-estimation of bee abundances, though it is worth highlighting that the relationship between bee abundances and residence levels in real landscapes is not necessarily linear, and instead is more likely to be the result of a variety of influences, including the frequency of floral replenishment.

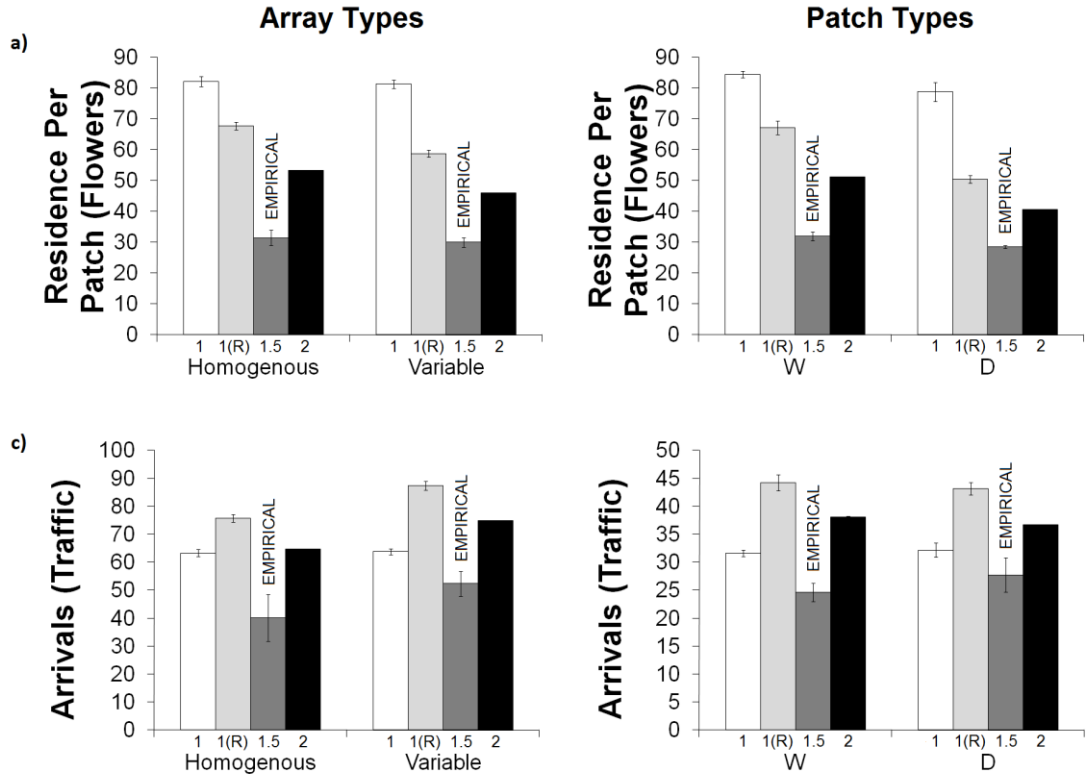

Figure 5.9. Mean patch residence (in flower visits) and arrival rates (y-axis measurements) of bees, comparing '1 Bee' (HARVEST simulation of one bee, with the same replenishment intervals as the '2' simulation), '1 (R) Bee' (HARVEST simulation of one bee, with replenishment intervals scaled accordingly to compensate for reduced simulated bee abundance), '1.5 Bees' (Empirical data) and '2 Bees' (Original HARVEST data with simulation of 2 bees). Comparisons shown between homogenous and variable landscape configurations, and W and D patches. Empirical residence data converted to flowers visited by assuming one flower visit for every 2 seconds of residence. For empirical data, averages taken across all patches of each type, and across all sessions. For model data, averages taken across all patches of each type, and all 10 trial replicates. Error bars indicate ± 1 standard error among sessions or trials.

5.4.2 Sensitivity analysis of results to variation in nectar replenishment rate

Whilst most of the model's parameterisation for this study came directly from the empirical observations, I had no direct measure of nectar replenishment rate and therefore estimated this parameter value. In the absence of direct measurements, I here outline a sensitivity analysis that I conducted to ascertain the potential impact of the replenishment frequency estimate being inaccurate.

I performed the sensitivity analysis with both homogenous and variable landscape scenarios. I tested replenishment rates that ranged from being twice as frequent as the original rates used to being twice as slow; specifically, I tested replenishment intervals for W patches ranging from 2,000 time units (200 seconds) to 8,000 time units (800 seconds) inclusive, in intervals of 1,000 time units. For variable landscape scenarios, the 2 : 1 D to W ratio was always maintained because we have empirical backing for this relationship, and so D patch replenishment frequency followed from the trialled W patch frequencies. All other parameter values were left identical to the main experiment.

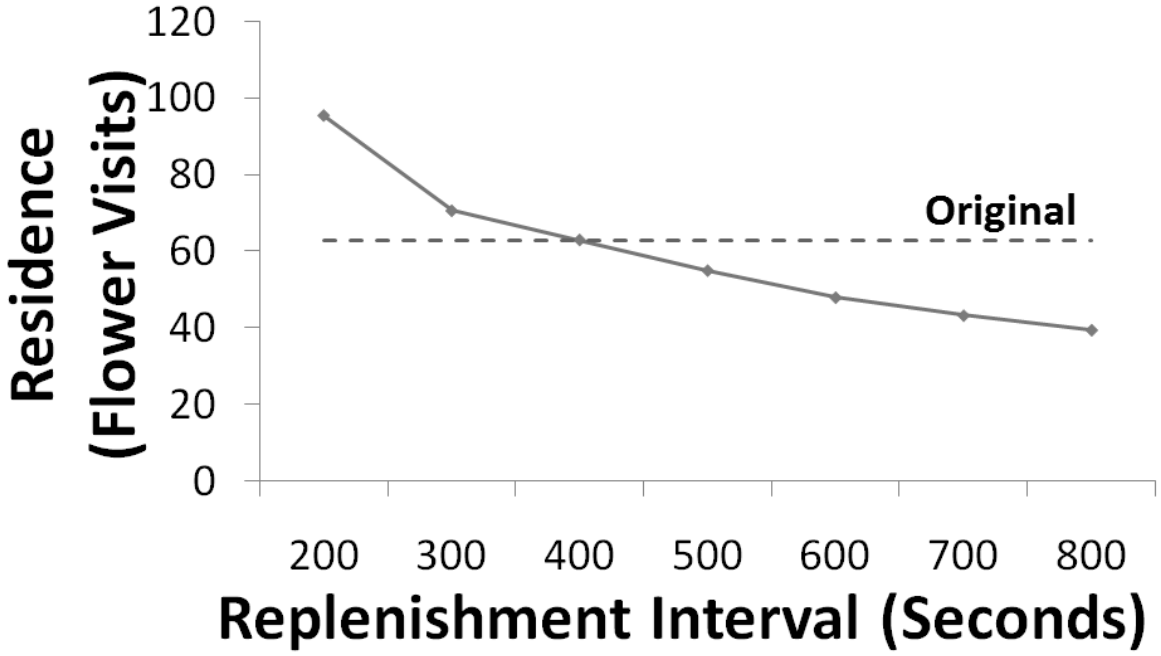

Figure 5.10. Mean patch residence per bee (y-axis) with varying replenishment intervals (x-axis) tested in a homogenous landscape system in the model, replicating the empirical array. Residence predicted from using the original replenishment interval is shown by the dotted line.

Patch residence is relatively sensitive to replenishment rate variation (figure 5.10). Deviations are greatest when replenishments are more frequent than my original estimate. However, whilst *B. napus* flowers have been shown to replenish within 30 minutes of being depleted (Meyerhoff, 1958 in Pierre *et al*, 1999), it is highly unlikely that patches replenished more quickly than once every 3 minutes and 20 seconds.

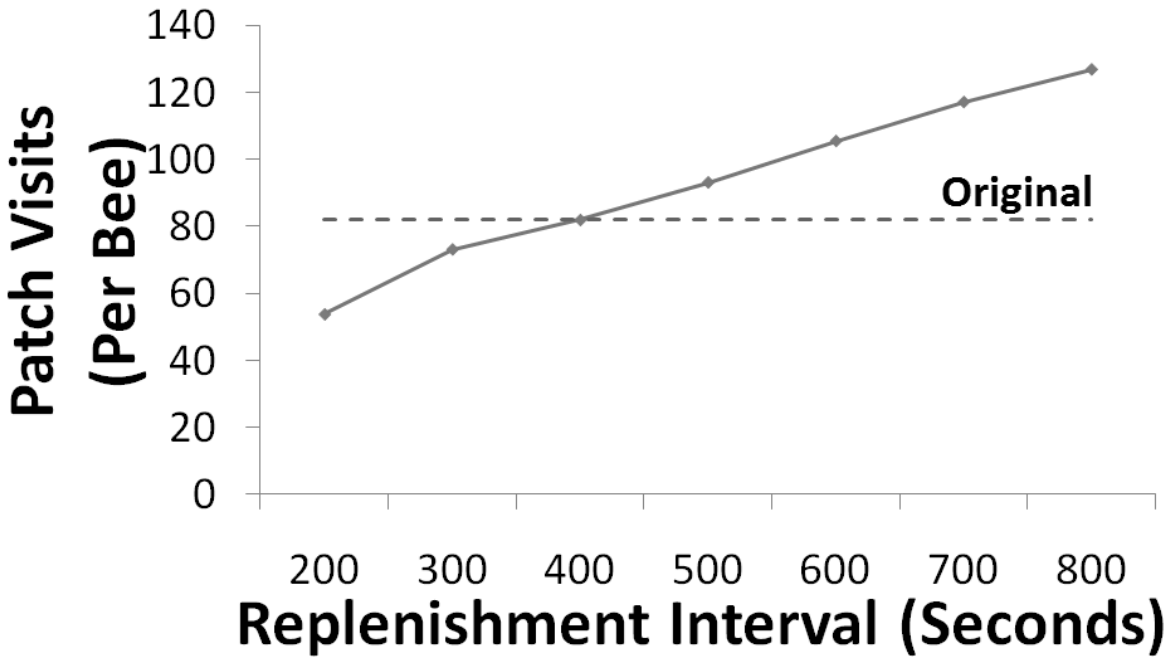

Fig 5.11. Mean toal patch visits per bee (arrival data - y-axis) with varying replenishment intervals (x-axis) tested in a homogenous landscape system in the model, replicating the empirical array. Number of patch visits predicted from using the original replenishment interval is shown by the dotted line.

Arrival data also is sensitive to replenishment rate variation (figure 5.11). However, assuming that that the rate of replenishment is accurate to within around 3 minutes, the predicted number of arrivals is not changed significantly.

5.5 Discussion

5.5.1 Insights into bee foraging behaviour

In qualitative terms, the high levels of traffic predicted by the model agree with my observations of real bees, as they also moved between patches frequently. It would, therefore, seem reasonable to hypothesise that the frequent inter-patch transitions exhibited by real bees are also *primarily* due to encounters with depleted flowers. This is consistent with the Threshold Departure Rule (Hodges, 1985b), which predicts that the likelihood of patch departure is dependent upon the volume of nectar sampled from flowers, and therefore an increased abundance of depleted flowers would increase the number of inter-patch transitions. Furthermore, the Threshold Departure Rule has been shown to be an accurate predictor of small-scale departure decisions made by bumble bees (Hodges, 1985a).

Indeed, it is possible that the nectar production rate of plants has been adapted to promote bee movements and thereby enhance cross-pollination (Mitchell, 1993; Pleasants and Chaplin, 1983; Real and Rathcke, 1991). Plants may have evolved to promote increased pollinator visitations that would increase pollination success in their populations. Consequently, the frequency of movement of pollinators in real foraging landscapes would be dependent upon the rate at which nectar is produced in the plant populations therein.

5.5.2 Prediction of observed foraging movements

Overall, HARVEST failed to achieve quantitative accuracy in terms of either residence or traffic predictions. I now discuss each of these components of behaviour in more detail.

5.5.2.1 Patch-to-patch movements, overall traffic (SWIPT) and traplining

Economically optimised HARVEST bees tended to move between patches in the grid frequently. This is likely because the rate at which flowers are visited (once every 2 seconds) far exceeds the rate of floral replenishment (once every 10 minutes). Consequently, bees are more likely to encounter flowers that have been emptied of nectar, either by themselves or their competitors, and the probability of a patch departure being triggered is increased. The sensitivity analysis that assessed the impact of replenishment rate variation on arrival rates into patches supports this theory, because it demonstrates that less frequent floral replenishments can significantly increase the level of bee movement within the system. Specifically, the over-estimates of traffic given by HARVEST would be eradicated if the rate of replenishment were doubled, though residence predictions would then become more inaccurate.

If flowers were to replenish instantly, inter-patch traffic would be abolished because bees would never encounter an empty flower, and so a bee's estimate of the quality of their current patch would never fall below that of an alternative. Consequently, the relatively poor match between observed and predicted levels of inter-patch traffic could be due to unrealistic estimate of nectar replenishment rates. I did not directly measure the nectar production rate of flowers in the grid. If I were to repeat the experiment, I would take an accurate measure of nectar production rates so that this could be achieved. This would be preferable to implementing 'tuned' replenishment values that replicate the empirical data, because biological plausibility would be reduced if there were no external validation for the calibration (Kleijnen, 1995). This would be particularly

important if continuous nectar distributions were implemented, rather than the binary floral reward scheme, and I discuss this potential extension in Chapter Seven.

5.5.2.2 Residence and arrival rates

HARVEST typically over-estimated the number of flowers that bees visited per patch, and the level of movement between patches. This could be because some of the parameterisation of the model may have itself been inaccurate. Specifically, as I discussed above, arrival rates and residence are sensitive to floral replenishment rates. I therefore may have inaccurately estimated nectar replenishment frequency.

Additionally, the higher residences predicted by the model may be an artefact from the implementation of a closed system, within which bees must forage until they have filled up to capacity. The grid in the empirical study did not represent a closed system, as bees were free to move into and out of the system at any point, and could forage at locations other than those on offer in my configuration. Indeed, given the relatively short real bee residences observed, it is highly unlikely that the bees were filling to capacity solely within the grid I had configured, as bees typically visit in the region of 500 flowers to fill to capacity (Cresswell *et al*, 2002). In the model's closed system, residences could be lengthened because bees are forced to learn the qualities of patches therein at a faster rate, thereby promoting more rapid patch fidelity behaviours. Additionally, the nature of the closed system means that all moves that are made are within the system itself, whereas the inter-patch transitions that bees made in the empirical grid would have only represented a proportion of their total inter-patch movements in a foraging bout. Therefore, the model's predictions of traffic levels may have been artificially high because bees were constrained to moving to another location in the grid when a patch departure occurred.

5.5.3 Traplining

HARVEST bees fail to trapline, and this may have contributed to inaccurate predictions of residence. The majority of moves made by real bees were orbits around the edges of the array. It is believed that bumble bees have a strong sense of directionality (Chittka *et al*, 1999; Wehner *et al*, 1996; Pyke and Cartar, 1992) and use landmarks when navigating (Goulson and Stout, 2001; Plowright and Galen, 1985; Birmingham and Winston, 2004; Colborn *et al*, 1999; Wehner *et al*, 1996). The edges of the experimental array might be advantageous locations to spot landmarks or may even represent landmarks in and of themselves (Plowright and Galen,

1985). In contrast, HARVEST bees tended to move between patches with no discernable pattern. This is unsurprising, as the model bees have no economic reason to forage systematically. Traplining behaviour could emerge if an economically-motivated forager is conscious of the economic advantage of revisiting a site after some time has elapsed since its last visit, because it believes the food supply there will have replenished (Thomson, 1996). The bees in the model do not assume that patches get better; when foraging, patch quality estimates, or action values, remain unchanged for patches other than the bee's current patch. Theoretically, a traplining mechanism could be introduced into the model by implementing a 'floating bottom' mechanism for the action-value calculation, in which the estimated qualities of alternative patches increase over time. I discuss this further in Chapter Seven.

5.6 HARVEST vs observations – overall comments

As in my study, others have found that quantitative model accuracy can be diminished when there are potential inaccuracies in parameterisation. Bernstein *et al* (1988) found that, in a depleting environment, forager distributions less closely matched the predicted Ideal Free Distribution when the rate at which a forager could consume resources was increased. Lima (1984) found that his model underestimated the number of holes that woodpeckers would sample for food, and speculated that the discrepancies may have been caused by erroneously assuming omniscience with regards to some environmental variables. Even models that appear to have good quantitative agreement with empirical data - such as the OFT model of Krebs *et al* (1978) that accurately predicts the number of hops that a great tit will make before committing to a feeding site – can be misleading, because the parameterisations can be artificially accurate if they are made post-hoc (Kamil, 1983).

An increase in the rate of floral replenishment would improve the accuracy of HARVEST's inter-patch traffic predictions, but this in turn would worsen the accuracy of the residence predictions. This occurs because HARVEST bees are constrained to foraging in a closed system, whereas real bees would be foraging in an open landscape, and would therefore be responding to a wider landscape than that simulated by the HARVEST experiments. Therefore, the relationship between residence and inter-patch traffic in the closed system will differ from the real landscape, because a decrease in residence in the closed system necessarily implies an increased level of traffic between the six patches in the matrix. In the real landscape, movements between the patches in the matrix would only represent a proportion of the total moves made by a bee in a foraging bout. Therefore, it may be beneficial to simulate a wider foraging landscape to assess potential improvements in HARVEST's predictive accuracy for

small-system experiments. Alternatively, the empirical investigation could be carried out in cage conditions (e.g. Robertson *et al*, 1999), to mimic the closed system simulated in HARVEST.

Arguably, it may have been advantageous to have selected a more isolated location for the empirical study, which was situated in close proximity to hedgerows, large patches of wild flowers and woodland areas. By increasing the isolation of the matrix, I may have extended the duration of visits of the bumble bees to our system, as they would have had to move further to find another suitable foraging location. This in turn may have increased patch residence levels, as the grid would have more closely approximated a closed system configuration. I showed in the previous chapter that bumble bees appear to exhibit this behaviour when foraging sites are more isolated. On the other hand, the presence of wild flowers and other foraging area around the site were likely attracting bumble bees into the grid, and it is possible that this area was already part of a larger 'trapline' of the bees I observed. In a more isolated grid, bumble bee abundances may have been lower, at least initially.

Despite the quantitative discrepancies with residence and arrival data, the qualitative results of this comparison between real bee behaviour and the predictions of the model are nevertheless not unencouraging for a programme aimed at simulating the landscape scale, because the mismatch can be attributed largely to difficulties in estimating floral replenishment rate, which is not a relevant variable at the landscape scale. Specifically, flower numbers are so vast and bees so sparse, that the depletion effect is minmal. I critically evaluate these implications in more detail in Chapter Seven. Meanwhile, with these caveats, in the next chapter I apply the model to the landscape-scale and generate gene-flow based predictions.

Chapter Six : Landscape-Scale Bee-Mediated Gene Flow and Containment

6.1 Introduction

The emergence of GM crops in agricultural systems necessitates the implementation of containment strategies that minimise the spread of GM genes into conventional crop varieties (Poppy and Wilkinson, 2005). Traditionally, proposed solutions either use separation distances (e.g. Damgaard and Kjellsson, 2005; Ingram, 2000; Ramsay *et al*, 2003; Colbach *et al*, 2009; Llewellyn *et al*, 2007) or barrier crop implementations (e.g. Morris *et al*, 1994; Messeguer *et al*, 2004; Reboud, 2003; Llewellyn and Fitt, 1996) to attempt to minimise the risk of GM to non-GM gene dispersal. Separation distance methods are based on the simple premise that gene flow levels are minimised as the distances between populations are increased (Cresswell, 2006). I discussed this pattern in Chapter Four, and demonstrated how HARVEST replicates this pattern from first principles of bee behaviour. Nonetheless, whilst some studies have demonstrated the superiority of separation distance strategies (e.g. Damgaard and Kjellsson, 2005), it has been shown elsewhere that the use of barrier crops – non-GM populations that are essentially 'sacrificed' to receive GM genes whilst protecting other non-GM populations – is a more effective approach for GM gene containment (Morris *et al*, 1994; Reboud, 2003). GM varieties of *B. napus* are a potential risk to crop purity, because their populations are susceptible to cross-pollination mechanisms which could hamper gene containment measures (Ingram, 2000).

Given that HARVEST is able to replicate the gene flow minimisation pattern obtained from increasing isolation distance, it is worthwhile to assess the model's predictions of the impact on GM to non-GM gene flow when barrier crop strategies are implemented. I have already shown in Chapters Four and Five that bees overwhelmingly tend to visit nearest neighbour resource sites when moving from one site to another, particularly at the economically unfavourable landscape-scale, where inter-site distances are significant. Therefore, in principle, HARVEST bees should be attracted to switch to barrier crops initially, instead of making direct moves between GM and target non-GM populations. To assess this, I propose that a potential means of containing gene flow would involve implementing a *Sacrificial Shield* between the target populations, which would be comprised of resource sites into which the level of gene flow is considered irrelevant or acceptable.

In this chapter, I undertake two main investigations. First, I assess how landscape-scale variability in reward abundance and landscape size potentially impact residence and inter-field traffic levels, and consequently theorise how gene flow levels could change according to these broad landscape characteristics. Second, I assess the potential efficacy of the barrier crop strategy, by using HARVEST to implement a series of landscape configurations that include a 'Sacrificial Shield'.

6.2 Parameterisation of the model for the landscape-scale

Up until now, the experiments I have conducted with the model have typically represented smaller patch-scale foraging scenarios. In this section I outline how the model was parameterised to capture larger-scale foraging landscapes where patches are considered to be fields, and the distances between them expressed at the kilometre-scale. I should highlight that the structure of the model remains unchanged when applied at this scale, however. This is important for two reasons. First, I have partially validated the rules of the model at the small-patch scale, which has given credence to the ability of the model to predict bumble bee foraging behaviours. If I were to alter the framework of the model, this reassurance would be lost, and any landscape-scale predictions would potentially lack credibility. Second, there is no reason to assume that bees' fundamental decision processes differ at larger scales, foragers would be experienced in responding to landscapes that offer variable size resource sites (Ricketts, 2001). The rules that I derived in designing the model were all based on first principles of bumble bee behaviour, taken from both previously observed bee behavioural phenomena and definitions of economically motivated foraging. Therefore, the rules that form the basis of the model should be scale-independent, since the economically motivated patch choice problem that the model attempts to solve is also scale-independent.

The most obvious difference when simulating landscape-scale scenarios is the size of the patches. At the smaller scale, we are effectively simulating small collections of flowers or even very small clumps of flowers. These smaller patches are common in the landscapes of pollinators and it is important to investigate them (Ricketts, 2001). However, at the landscape-scale, patches of flowers represent large fields of resource, such as *B. napus* fields. Intensification of agricultural practices and rising demand have both led to increased crop production rates over the last 40 to 50 years (Oerke and Dehne, 1997), and consequently crop fields can be significant in scale (Roschewitz *et al*, 2005). To capture this scale, I typically implement patch sizes of 100,000 flowers when applying the model at the landscape-scale. There are approximately 1,150 flowers per m^2 in winter sown *B. napus* (Hayter and Cresswell,

2006), so my patch size emulates a $100m^2$ field. Whilst a typical field of *B. napus* is approximately $326m^2$ (Wilkinson *et al*, 2003), the difference in size is irrelevant for HARVEST, as depletion and replenishment effects are not simulated at the landscape-scale (see below), and therefore only the proportion of full flowers in a field (*Q(i)*) can potentially affect foraging behaviour.

It is unlikely that one would find any real world fields that are either completely devoid of or replete with nectar, as plants can vary the rate at which they replenish nectar supplies according to how often their nectar supplies are depleted (Castellanos *et al*, 2002). I therefore set patch qualities at 0.5 for landscape-scale experiments, unless otherwise stated. At the kind of scale being simulated when exploring landscape-scale dynamics, the foraging impact of bees on standing crop levels is likely to be negligible, because the visitation rate to flowers is low (Hoyle *et al*, 2007). Therefore, I do not simulate depletion and replenishment mechanisms when using the model at this scale.

When simulating at the smaller scale, the distances between patches are often negligible, especially given the flight speed of bumble bees (Osborne *et al*, 1999). Small patch-scale scenarios were therefore often implemented with only minimal inter-patch distances. At the landscape-scale, patch choice decisions would take place over a much larger foraging radius, and in the case of bumble bees we know that this extends over the kilometre-scale (Osborne *et al*, 1999; Dramstad, 1996; Saville *et al*, 1997; Knight *et al*, 2005; Greenleaf *et al*, 2007; Goulson and Stout, 2001; Darvill *et al*, 2004). To ensure that these aspects are captured, I specified an inter-patch distance of 1km between cardinal neighbour patches at the landscape-scale, unless otherwise stated. Expressed as a flight time - assuming that a bee flies at a constant speed of 7 metres per second (Osborne *et al*, 1999) and that a single time unit in the model represents 3 seconds of real world time (Cresswell, 1999) - this translates to approximately 45 time units.

Each landscape-scale experiment simulated 7 bees foraging simultaneously. Whilst the number of bees is somewhat inconsequential when depletion and replenishment mechanisms are not simulated, because bee behaviour cannot be affected by the actions of other foraging bees, I maintained a colony size greater than unity to increase the number of trial replicates. Additionally, for each experiment, I ran 20 replicated trials, and in each trial I allowed bees to forage for 8 hours. All other parameterisations were as per discussions in previous chapters, and are summarised in Table 6.1.

Parameter	Identifier	Value
L	Number of patches in landscape	Variable
t(a, b)	Flight time between cardinal neighbours patch a and patch b	135 seconds
Q	Initial quality of patches	50%
F_{all}	Number of flowers in each patch	100,000
I_{all}	Replenishment interval for each patch	No replenishment
B	Number of bees in the grid	7
C	Bee's nectar carrying capacity	500 full flowers
h	Flower handling time	3 seconds
β	Sensitivity to individual flower sample	0.05

Table 6.1. Summary of general parameterisation for landscape-scale experiments

6.3 Investigation (i) : Landscape-scale behavioural exploration

6.3.1 Methods

6.3.1.1 Experiment LS1.1

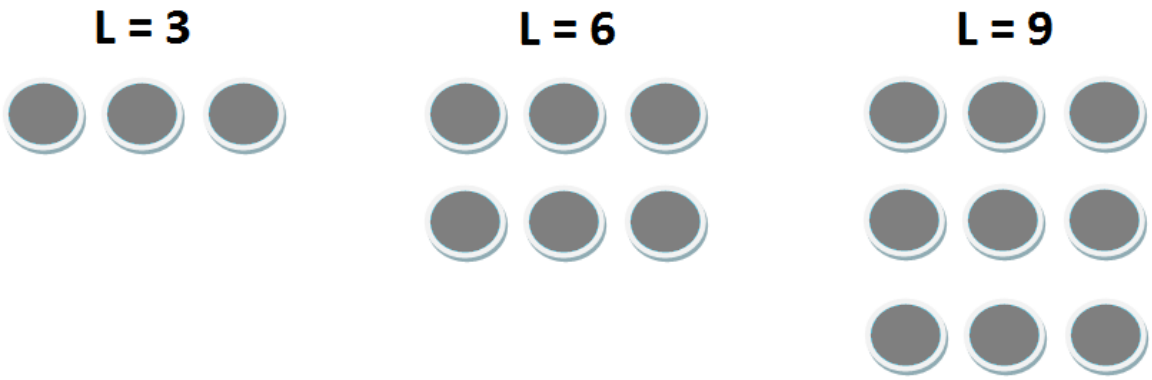

Figure 6.1. Example landscape configurations showing how landscape size (L) is increased to investigate landscape-scale dynamics; expansions to the landscape are made by adding extra rows to the matrix. Filled circles indicate fields.

In the first experiment I explored the impact of landscape size on foraging behaviour. Landscape size refers to the number of patches in the landscape (*L*), which at this scale is the number of fields of flowers available to the foraging bees. I tested values of *L* ranging from three to 24 fields, in intervals of three fields, giving a total of eight trial sets. Landscapes containing fewer than three fields are of no particular interest to my study as it is unlikely that bumble bees would have fewer than three large foraging sites to consider at any one time, given their large foraging radius (Osborne *et al*, 1999). I arranged fields in a matrix with three columns, so that every subsequent expansion to the size of the landscape added a new row of three fields to the bottom of the matrix (figure 6.1). All fields had a quality of 0.5, which remained constant throughout a trial due to the lack of depletion and replenishment simulation.

6.3.1.2 Experiment LS1.2

In the second experiment I altered the variability of the field qualities in the landscape to ascertain the patterns of foraging behaviour that would emerge. In this experiment, I trialled ten different degrees of variability - measured in terms of the range between the highest and lowest quality fields - with ranges tested from 0.1 to 1.0 inclusive, in intervals of 0.1. I did not trial a range of 0.0 as this was tested in Experiment LS1.1, in which a homogenous resource landscape

is implemented. All field qualities, except for those of the highest and lowest rated fields, were generated randomly for each trial of each trial set.

To ensure that the effect of varying landscape-size did not influence the results, I kept the number of fields in the landscape constant at six fields for all trials. I arranged the six patches in a matrix of two rows and three columns, thereby ensuring that the matrix was as close to a square as possible given the number of fields used, so as to minimise potential bias from particular spatial configurations.

6.3.2 Results

6.3.2.1 Experiment LS1.1

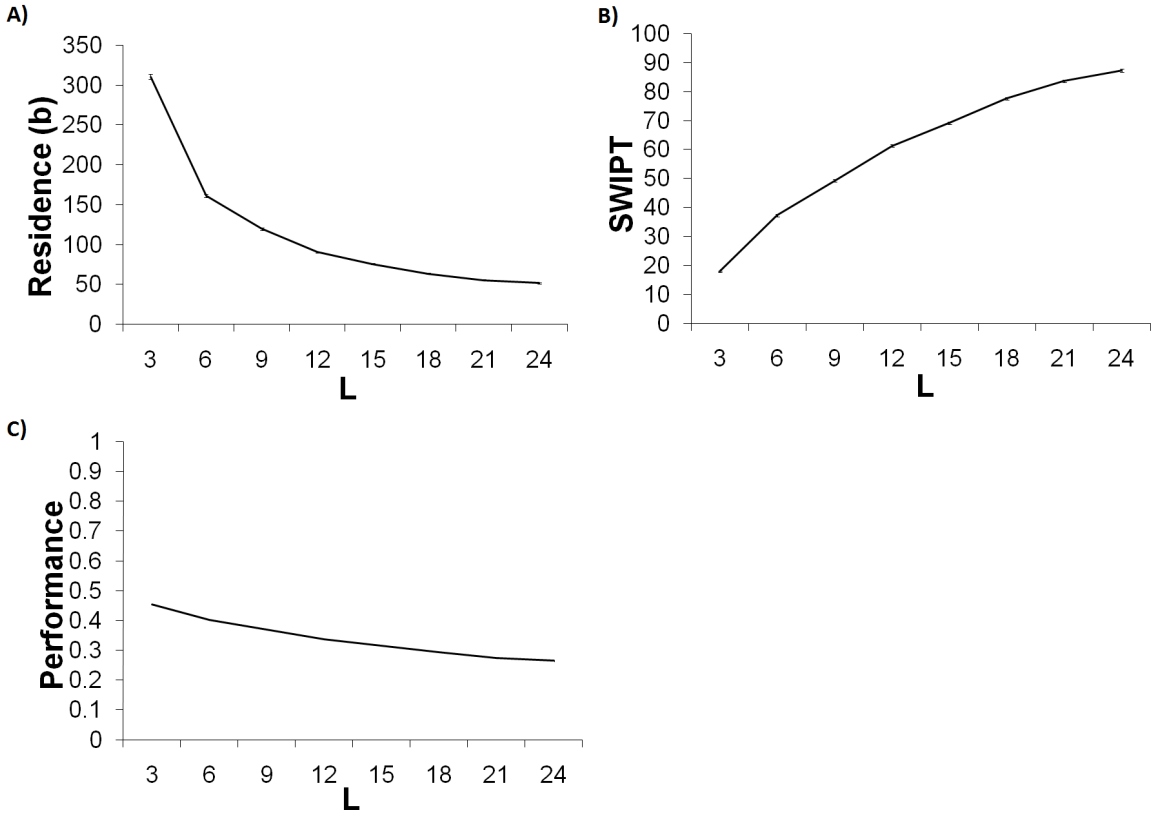

Figure 6.2. a) The mean number of flowers visited per bee per field (y-axis), with varying landscape size L (x-axis). b) System-wide inter-patch travel (SWIPT; mean number of inter-patch movements per bee per 8 hours – y-axis) with varying landscape size L (x-axis). c) Mean foraging efficiency per bee per foraging bout (y-axis), with varying landscape size L (x-axis). 7 bees simulated foraging simultaneously, 20 trial replicates per landscape size. In all cases, error bars show ± one Standard Error among trials.

A bee's average residence per field decreases significantly as the number of fields in the landscape increases (figure 6.2a). This decrease also exhibits an obvious and fairly rapid deceleration. Inter-patch traffic levels show a saturating increase as the number of fields in the landscape is increased (figure 6.2b). Foraging efficiency shows a relatively minor decrease as the size of a landscape is increased (figure 6.2c).

6.3.2.2 Experiment LS1.2

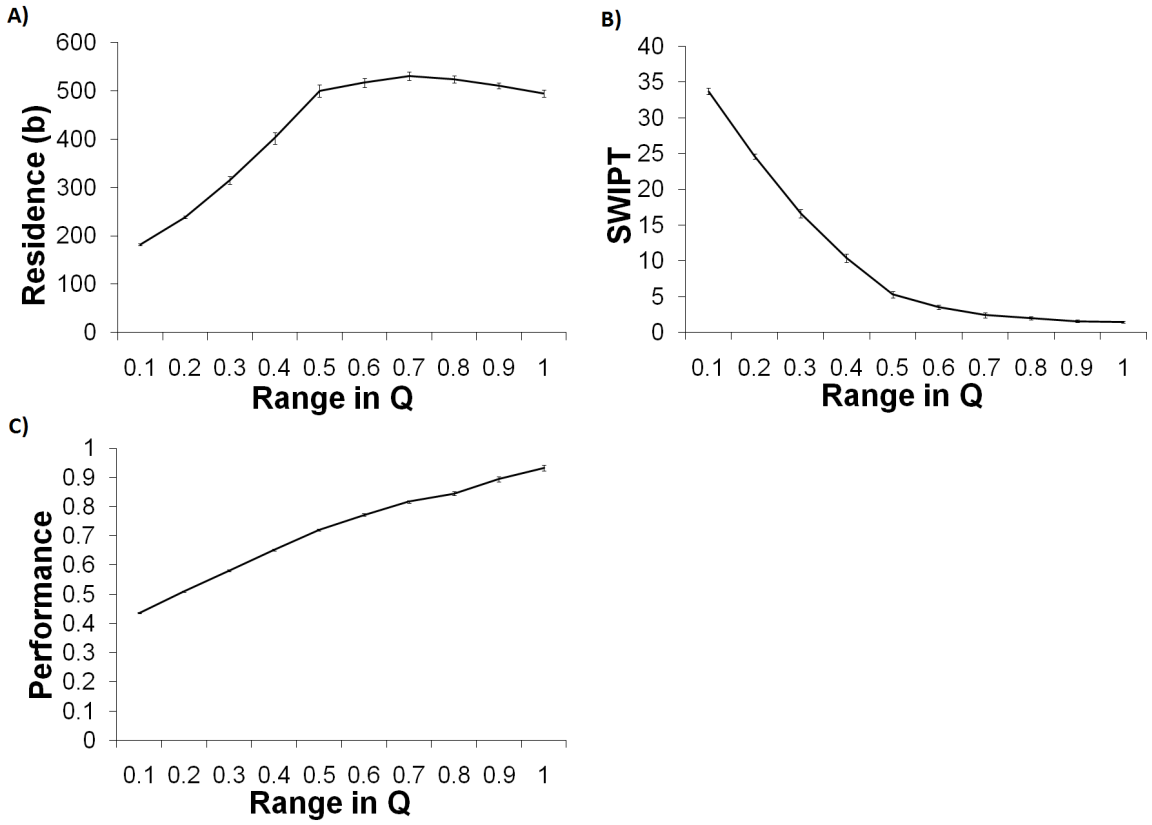

Figure 6.3. a) The mean number of flowers visited per bee per field (y-axis), with varying range in field qualities, Q (x-axis). b) System-wide inter-patch travel (SWIPT; mean number of inter-patch movements per bee per 8 hours – y-axis) with varying range in field qualities (x-axis). c) Mean foraging efficiency per bee per foraging bout (y-axis), with varying range in field qualities (x-axis). 7 bees simulated foraging simultaneously in a landscape of six fields, and 20 trial replicates per patch quality range. In all cases, error bars show ± one Standard Error among trials.

Average field residence exhibits a saturating increase as the reward availability in the foraging landscape becomes more variable, with a minor decrease observed with very high levels of variability (figure 6.3a). Residence tends to increase initially very sharply as variability is increased. Traffic levels showed a decelerating decrease with increasing reward variability. Foraging efficiency increasead linearly as landscape variability was increased (figure 6.3c).

6.3.3 Discussion

The main findings of these experiments are: (a) bees make more frequent field-to-field journeys when resource variability among fields is lower, and (b) bees make more frequent field-to-field journeys when the landscape contains a greater number of patches, or fields of flowers.

6.3.3.1 Finding (a) : why does SWIPT increase with decreasing variability in Q?

When a landscape is more variable, it contains fields of superior forage that exceed the foraging quality found in a homogenous landscape of half-full fields. A superior quality field is more likely to retain the attention of a foraging bee, as the probability of encountering an empty flower is reduced. Consequently the probability of a bee experiencing a sufficient 'run of bad luck' to instigate an inter-field move is also reduced. This is further compounded by the inferior quality of many of the other fields. As variability increases, the difference between the highest and lowest quality fields in the landscape is increased, and this effect becomes more pronounced.

Saturation in the increase in residence with increased variability occurs for two reasons. First, there is an implicit maximum retention point, beyond which the added superiority of the field has only a trivial impact on a bee's probability of moving away. In other words, the superior fields are of such high quality that improvements to their quality make little difference to site fidelity behaviour. Second, a bee has a fixed nectar carrying capacity and so only needs to collect a finite quantity of nectar before it finishes its foraging bout. For this experiment I set this capacity at 500 nectar units, and it is interesting to observe that a very sudden saturation occurs when residence increases to around 500 flowers.

The minor decrease in residence at very high levels of variability occurs because, in such scenarios, there are fields that are near-replete or completely replete with full flowers. The probability of encountering an empty flower in such fields is therefore marginal to nil. As a

result, bees foraging in these fields can fill to capacity at near maximum efficiency; that is, by sampling only 500 flowers. Average residence in such fields would therefore be marginally lower than in fields whose qualities were very high but lower than near-perfect, as bees sampling in the superior fields of such landscapes would still encounter a proportion of empty flowers.

There is no minor increase in traffic levels in very highly variable landscapes that mirrors the decrease in residence in such landscapes. This is because the ability of a bee to fill to capacity using fewer flowers in the field does not alter inter-patch traffic, as bees that have filled to capacity finishes their foraging bout. Therefore, the decrease in residence does not directly lead to an increase in inter-patch movement in this case.

Since more variable landscapes contain superior highest quality fields, foraging efficiency is increased as the proportion of a bee's time spent extracting nectar from full flowers is increased. The increase is near linear because the increase in the quality of the best fields in the landscape is linear as landscapes become more variable. The strength of the linearity demonstrates that the model bees are very strong learners, as they are able to identify and settle in these superior fields with notable rapidity, improving their foraging efficiency in line with the increases in foraging efficiency potential in the landscape.

6.3.3.2 Finding (b) : why does SWIPT increase with increasing L?

If there are more available fields, there is a greater probability that a move to an alternative field will be considered preferable to staying in the current field, given that the last perception of each field is likely to vary, at least slightly. With increased probability of moving away from a current field, the average residence per field will be shorter.

The decrease in residence with increasing landscape size is not linear, but rather saturates. This occurs because there is effectively a minimum residence when a bee visits a field, which is determined by β and the quality of the field. Since β determines the bee's sensitivity to individual flower encounters, the value of β implemented will determine how many poor flowers are required as a minimum to instigate a move away from the patch, all else being equal. Even assuming a bee encounters nothing but empty flowers in its chosen field from its

time of entry, it will have to visit this minimum 'quota' before its estimate of the field would fall low enough to instigate an inter-patch move.

Foraging efficiency is lower in larger landscapes because they yield increased inter-patch traffic levels. In a homogenous landscape, inter-patch movements are never beneficial, because a bee can gain no advantage from moving to an alternative foraging site, and movements will therefore yield penalties without gain. Being incompletely informed foragers, however, the bees are not aware of this and will move to other fields due to runs of bad luck in their current field. The decrease in foraging efficiency is relatively minor, however, because all patches have adequate resources.

6.3.3.3 Consequences for gene flow

The findings of these experiments imply that gene flow levels will be higher when (a) resource variability among fields is lower and (b) when the landscape contains a greater number of patches, or fields of flowers. All else being equal, the E-Psi-b model (Cresswell *et al*, 2002) tells us that an increase in sink patch residence will decrease the level of pollinator-mediated gene flow from a source into that sink. Similarly, a decrease in movement within a foraging system decreases the probability of a source-to-sink transition occurring, which also reduces the predicted level of pollinator-mediated gene flow. The experiments I have presented here could help characterise the types of landscape that could promote minimisation of gene flow activity.

Since longer residences and fewer inter-patch moves are optimal for gene flow minimisation, an optimal landscape should contain fewer available fields. In the real world, limiting the availability of foraging sites may not be practical given the large foraging radius of bumble bees (Osborne *et al*, 1999) and the high resolution at which the landscape would need to be considered, as smaller scale patches would need to be restricted too, since they could also promote exploration (Ricketts, 2001). However, agricultural configuration strategies that minimise the density of large-scale foraging sites could result in a reduction in bumble bee mediated gene flow levels. Smaller densities would increase inter-site distances which tend to minimise gene flow levels (Crane and Mather, 1943; Rieger *et al*, 2002), and could potentially lower the probability of bees finding alternative sites in their foraging bouts.

Experiment LS1.2 showed that bees are less inclined to move when there are foraging sites that can provide plentiful supplies of nectar. This pattern is in line with previous findings that show that bumble bees are typically risk-averse (Harder and Real, 1987; Cartar, 1991; Cartar and Abrahams, 1996) and powerful learners that can identify superior foraging options (Pleasants, 1981; Ott *et al*, 1985; Cartar, 1991; Dall *et al*, 2005; Robertson *et al*, 1999). Whilst fine-scale manipulation of variability may be difficult to achieve, especially at the landscape-scale, it is plausible that gene flow minimisation strategies could seek to design foraging landscapes that contain a mix of both extremely abundant nectar sites and sites devoid of food supply. The predictions of the model suggest that such landscape configurations could potentially reduce the influence of bee-mediated gene flow processes.

6.4 Investigation (ii) : The Sacrificial Shield as a strategy for gene containment

6.4.1 Introduction to Investigation (ii)

Consider a scenario in which two bordering counties in the UK are to grow varieties of *B. napus*. Assume that one county has been designated to grow only GM varieties of the crop, whilst the other will only grow conventional varieties. This scenario is obviously a simplification, and potentially brings with it a variety of practical and political issues (Bullock and Desquilbet, 2002), but for our purposes we can think of this as representing any scenario in which GM and non-GM growing zones are identified. The county growing GM crops is designated as the *GM County*, whilst the other county is the *Conventional County*. We wish to minimise the levels of bee-mediated gene flow from fields in the GM County (the source fields) to the Conventional County (the sink fields). In an attempt to achieve this, a configuration of conventional fields that border the counties is set aside by one or other of the counties (or both). The owners of these fields are prepared to tolerate higher than threshold levels of GM seed set within them, and so they form the Sacrificial Shield. All other Conventional fields must retain their purity; that is, they must contain no more than a determined threshold of GM seed set.

To explore the feasibility of this containment strategy when applied to landscape-scale bumble bee foraging dynamics, and to ascertain how the shield and the counties should be configured to maximise the efficacy of the strategy, I conducted a set of four experiments using the model at the landscape-scale. I tested how the implementation of a shield affected predicted levels of gene flow from sources to sinks.

6.4.2 Methods

6.4.2.1 General Methods

Each county was comprised of nine individual fields, organised into a square matrix of three rows and three columns. Cardinal neighbours within each matrix were separated by a distance of 1km (45 time units), with all other distances calculated geometrically from this. When the sacrificial shield was implemented, it was comprised of three fields. In experiments LS2.2, LS2.3 and LS2.4 the shield was positioned to sit as a direct spatial barrier between the two counties, in a single column of three fields. In experiment LS2.1, the position of the shield was varied, and was sometimes placed horizontally so as to form a row of three fields. The nest was trivially equidistant (1 time unit flight time) from all fields in the landscape for all experiments. Each field in the landscape, whether part of a county or part of the shield, contained 100,000 flowers. In experiments LS2.1, LS2.2 and LS2.3 each field's quality was exactly 0.5 and remained constant throughout the experiment as the depletion and replenishment of nectar was not simulated. In experiment LS2.4 the quality of county fields remained constant at 0.5 but the quality of the shield fields was varied.

6.4.2.2 Experiment LS2.1

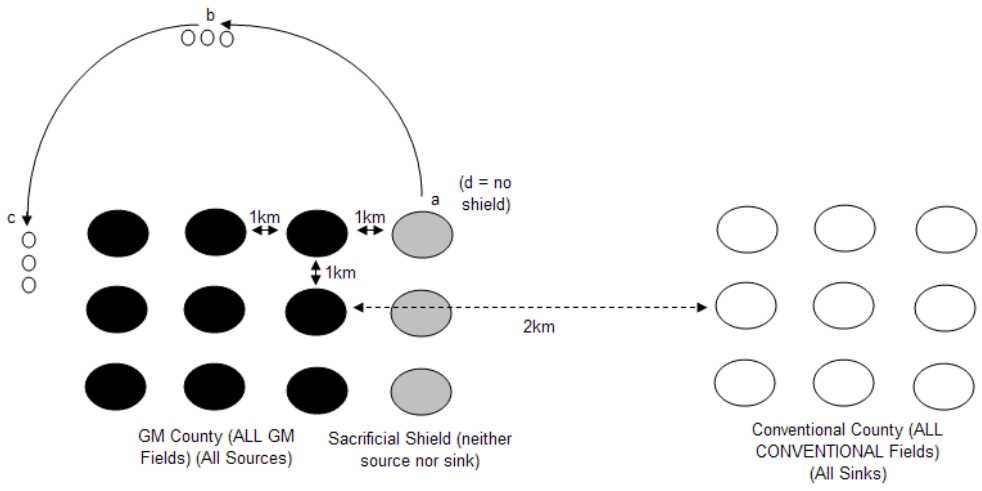

Figure 6.4. Configuration of the landscape and the shield in Sacrificial Shield Experiment LS2.1. GM fields shown as black circles, shield fields as grey circles and conventional fields as white circles. In configuration (a) the shield is placed directly between the two counties. In configuration (b) the shield is placed to the north of the GM county. In configuration (c) the shield is placed to the west of the GM county (on the opposite side to the county barrier). In configuration (d) no shield is used. In all cases, the counties are separated by a distance of 2km.

In experiment LS2.1 I tested how the presence of a sacrificial shield affected predicted levels of gene flow from the GM County fields to the Conventional County fields, and the effects of positioning the field at different orientations with respect to the GM County (figure 6.4). I ran four trial sets, and each trial set was named according to a grading scheme that described how closely the sacrificial shield resembled a direct spatial barrier between the two counties. In the *Grade 3* trial set I placed the sacrificial shield directly in between the two counties, arranged as a column of three fields, and 1km from the nearest neighbours in each of the two counties. In the *Grade 2* trial set I placed the shield in a row of three fields at the top of the GM County. Whilst this removed the direct barrier between the counties, the barrier could potentially act as an attractor to retain foraging attention in the GM County and minimise moves away from it. In the *Grade 1* trial set I placed the shield in a column of three fields on the opposite side to the border between the two counties. The principle being tested here was effectively the same as that of the *Grade 2* trial set, but this configuration is graded lower as it is further from the border between the counties, and is therefore less likely to directly divert attention away from inter-county movements for bees who are foraging in the border fields of the GM county. In the *Grade 0* trial set, I completely removed the shield from the landscape.

6.4.2.3 Experiment LS2.2

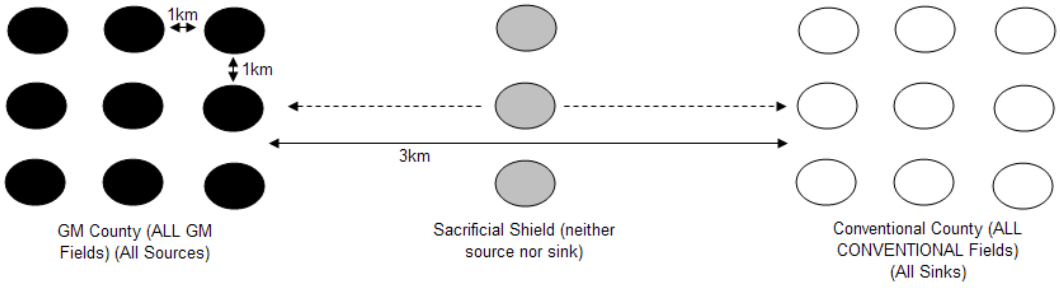

Figure 6.5. Configuration of the landscape and the shield in Sacrificial Shield Experiment LS2.2. GM fields shown as black circles, shield fields as grey circles and conventional fields as white circles. The shield is placed at varying intervals in the space between the two counties. In all cases, the counties are separated by a distance of 3km.

In experiment LS2.2 I tested how the effects obtained from isolation and increasing inter-patch distance could be brought about by positioning the shield at different intervals within the separation space between the two counties (figure 6.5). In experiment LS2.1 I had assumed that the separation space between the counties was 2km (excluding spatial dimensions of the fields

which are discounted in the model). To allow for greater scope for testing in this experiment, I increased the distance between the counties so that they were separated by a distance of 3km. I tested the effects of placing the shield at seven different intervals within the county separation space, and ran a trial set for each variation. The shield was placed at the following distances from the GM County : 1 time unit (i.e. – trivial distance, or 21m), 0.5km, 1km, 1.5km, 2km and 2.5km. I also tested the placement of the shield 1 time unit from the Conventional County. As the shield was placed further from the GM County, it was moved nearer to the Conventional County, much like the peaked isolation experiment I described in Chapter Four.

6.4.2.4 Experiment LS2.3

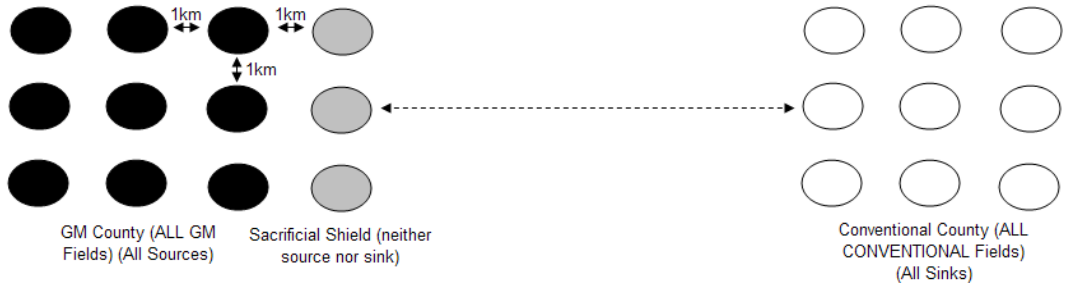

Figure 6.6. Configuration of the landscape and the shield in Sacrificial Shield Experiment LS2.3. The counties are separated by varying distances. In all cases, the shield is fixed to be 1km from the border of the GM county. GM fields shown as black circles, shield fields as grey circles and conventional fields as white circles.

In experiment LS2.3 I again manipulated separation distances within the model to test how this affected gene flow, but this time I altered the separation distance between the counties whilst holding the position of the shield at a constant distance of 1km from the GM County (figure 6.6). In real world terms, this could equate to growing border fields further inside the counties. For simplicity, I defined the shield as being part of the GM County, and therefore expressed inter-county distances in terms of the distance between the shield and the Conventional County. I tested separation distances of 0.25km, 0.5km, 1km, 1.5km and 2km.

6.4.2.5 Experiment LS2.4

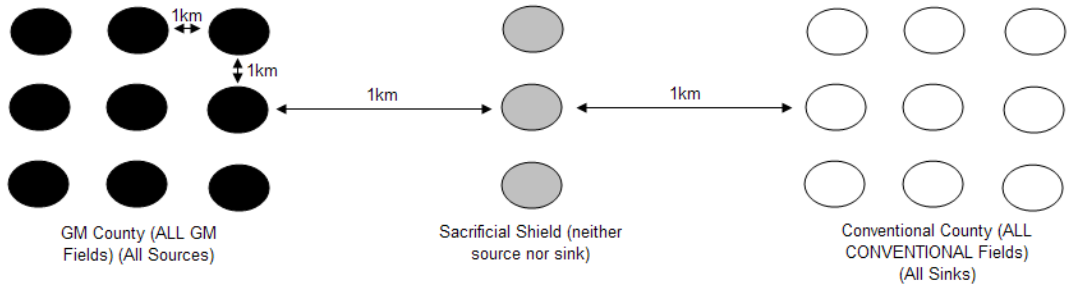

Figure 6.7. Configuration of the landscape and the shield in Sacrificial Shield Experiment LS2.4. The qualities of the fields in the shield are varied. In all cases, the shield is fixed to be 1km from the border of the GM county and 1km from the border of the Conventional county. GM fields shown as black circles, shield fields as grey circles and conventional fields as white circles.

In experiment LS2.4, I manipulated levels of nectar availability in the shield to assess this potential influence. We already know from my previous experiments that the availability of reward in foraging sites affects residence and inter-patch traffic levels of visiting bees. Clearly, this could potentially alter the probability of bee-mediated gene flow events from the GM County to the Conventional County. To minimise any potential spatial bias, I placed the shield directly in between the two counties and equidistant from each, with a separation distance of 1km (figure 6.7). I also ensured that all three fields in the shield were of the same quality in any given trial set. I tested shield field qualities of 0.1, 0.3, 0.5, 0.7 and 0.9. I avoided testing the extreme qualities of 0.0 and 1.0 because such scenarios are unrealistic.

6.4.3 Results

6.4.3.1 Experiment LS2.1

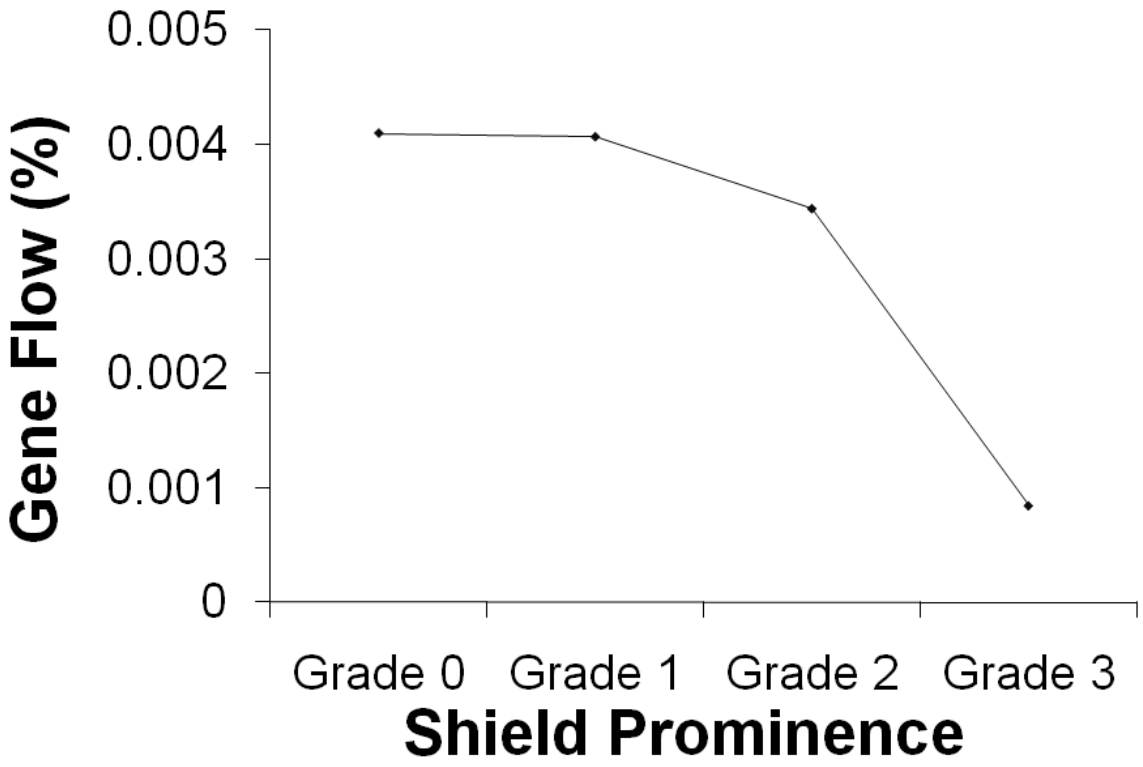

Figure 6.8. Gene flow, expressed as the percentage of GM pollen deposited by bees in the Conventional county (y-axis), with varying grades of prominence of the shield (x-axis). Grade 3 is the most prominent, with the shield placed as a direct barrier between the counties. Grade 0 is the least prominent, with no shield present in the landscape. Points are interpolated for ease of observation only. 7 bees were simulated over 20 replicated trials.

The presence of the Sacrificial Shield within the foraging landscape has a clear effect on the level of bee-mediated gene flow from the GM County to the Conventional County (figure 6.8). When the shield is present and most resembling a barrier between the two counties (Grade 3), the level of gene flow from source to sink is less than 25% of that predicted when there is no shield present. When the shield is present but not positioned directly between the counties, the level of gene flow is lower than that predicted without the shield, but not to the extent that the Grade 3 barrier provides. When positioned to the north of the GM county, the level of gene flow is approximately 83% of that found in the shield-less scenario. When the shield is placed on the opposite side of the border between the counties, the predicted level of gene flow is almost identical to the scenario where the shield is absent. In all cases, the level of bee-

mediated gene flow from the GM County to the Conventional County is predicted to be very low.

6.4.3.2 Experiment LS2.2

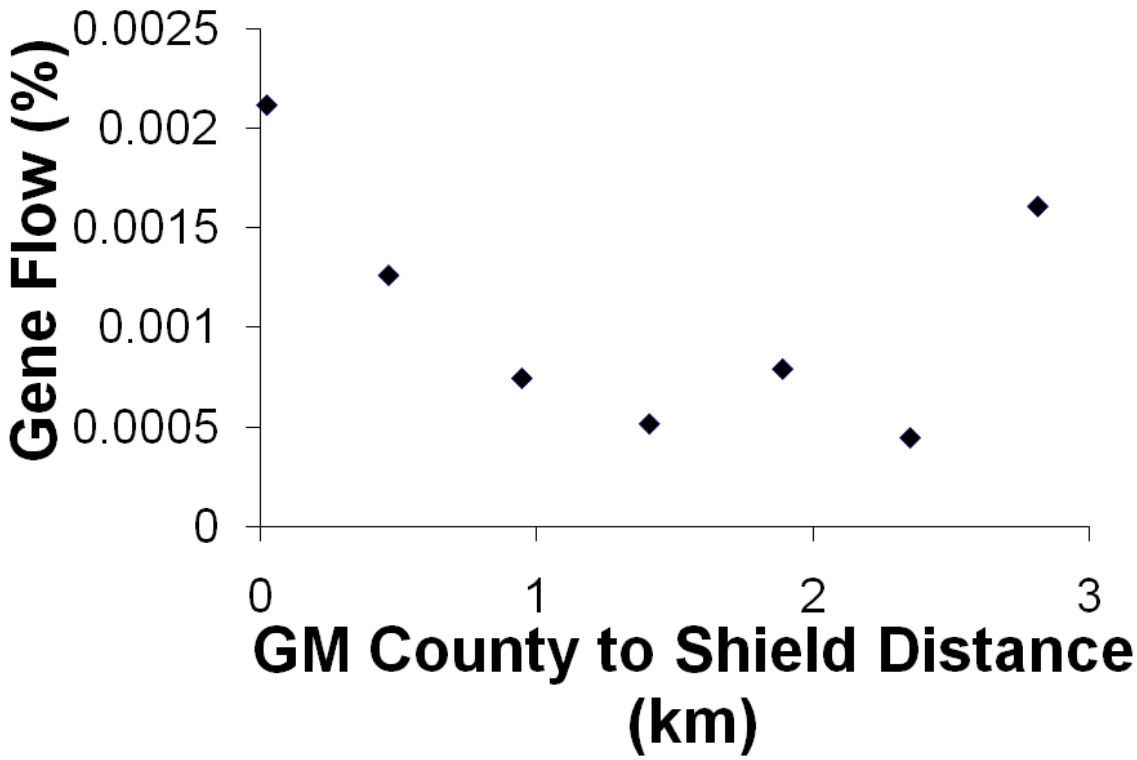

Figure 6.9. Gene flow, expressed as the percentage of GM pollen deposited by bees in the Conventional county (y-axis), with varying distance of the shield from the GM county (x-axis). Increased distance from the GM county implies decreased distance from the Conventional county. 7 bees were simulated over 20 replicated trials.

The level of gene flow from the GM County to the Conventional County is higher when the shield is closer to either of the counties (figure 6.9). The level of gene flow is generally much lower when the shield is positioned centrally, and nearer to equidistance from both source and sink counties. As the shield is moved further from the GM County and closer to the Conventional County, a near-parabolic curve is observed in gene flow levels.

6.4.3.3 Experiment LS2.3

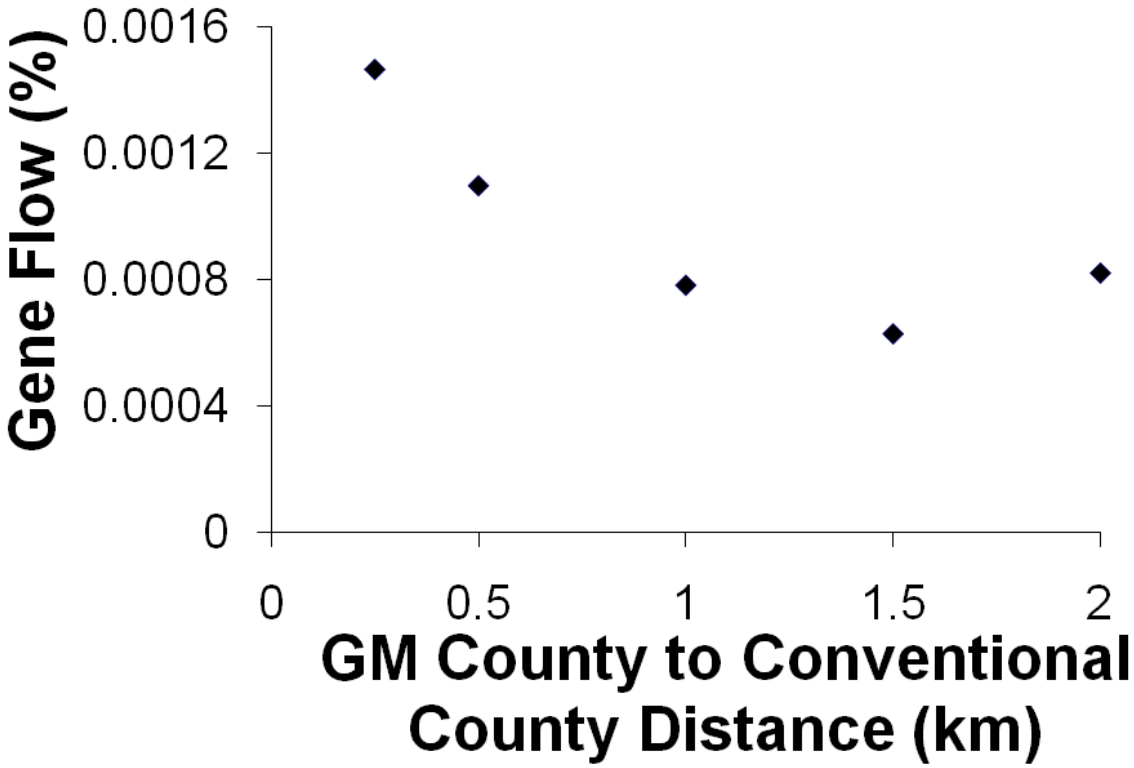

Figure 6.10. Gene flow, expressed as the percentage of GM pollen deposited by bees in the Conventional county (y-axis), with varying distance between the GM and Conventional counties (x-axis). 7 bees were simulated over 20 replicated trials.

When the counties are closer together, the level of gene flow from source to sink county is higher (figure 6.10).

6.4.3.4 Experiment LS2.4

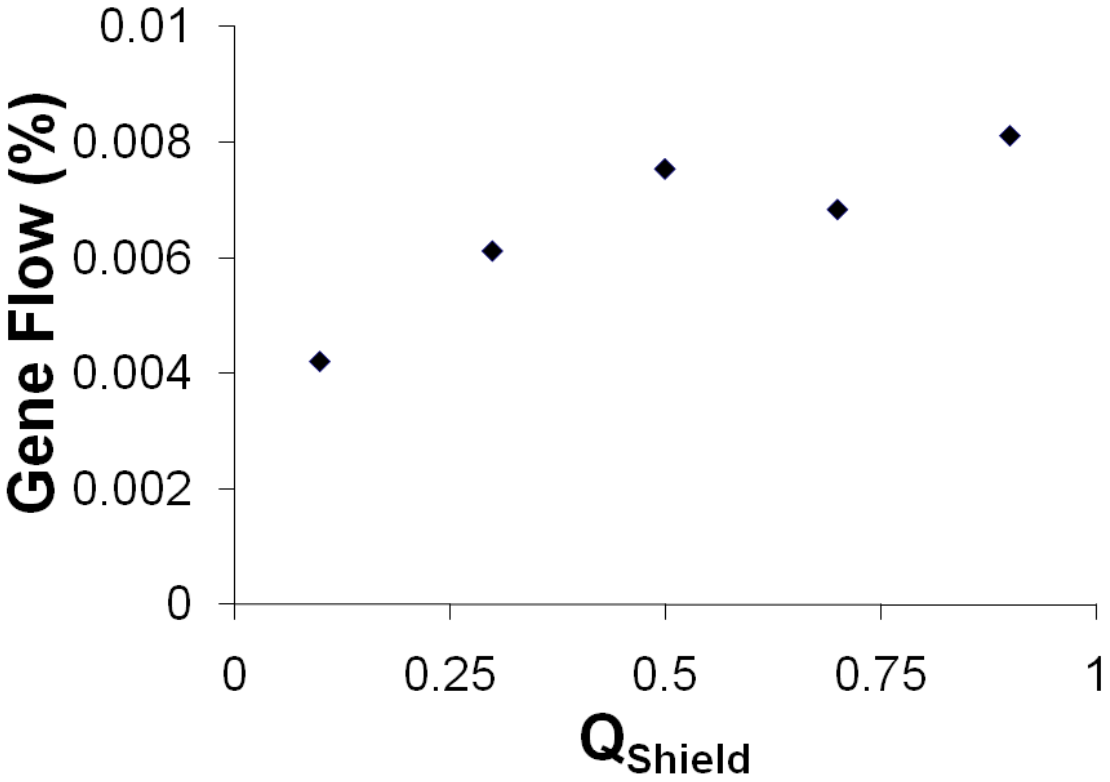

Figure 6.11. Gene flow, expressed as the percentage of GM pollen deposited by bees in the Conventional county (y-axis), with varying quality of the fields in the shield (x-axis). 7 bees were simulated over 20 replicated trials.

The level of gene flow from the GM County to the Conventional County increases as the quality of the fields in the Sacrificial Shield increases (figure 6.11). The increase decelerates with higher shield qualities.

6.4.4 Discussion

I can identify the best shield strategy for containment as the configuration of a series of barren fields directly between source and sink populations, and equidistant from both. Additionally, if the internal configurations of the populations can be altered, or if the populations can be moved, the populations should be configured so as to be separated from each other as much as possible. However, choice is somewhat inconsequential, because an obvious result from this landscape-scale exploration is that the level of bumble-bee mediated gene flow is predicted to be very low in realistic landscapes containing many foraging fields separated by non-trivial distances. All of the gene flow predictions presented here are far below EU Directive Thresholds for determining

crop purity, in which no more than 0.9% of GM seed set in a conventional variety population must be found to avoid classification of the crop as a Genetically Modified Organism (CEC, 2003). Such low levels of gene flow are predicted primarily because of the low levels of inter-patch traffic and long patch residences predicted in kilometre-scale landscapes comprised of fields of hundreds of thousands of flowers. From the E-Psi-b model, we know that low levels of traffic minimise the risk of source-to-sink transitions by pollinators, and longer residences increase the dilution of extrinsic seed set (Cresswell *et al*, 2002).

Clearly, real world landscapes do not solely offer foraging sites in the form of significantly separated fields. Foraging landscapes also contain many smaller patches of resource that need to be considered (Ricketts, 2001); indeed, my own empirical study that I outlined in Chapter Five demonstrated the attractiveness of very small-scale foraging sites for bumble bees. In the landscape size manipulation experiment that I outlined earlier in this chapter, I showed that an increase in the number of foraging sites in a foraging landscape increases levels of inter-patch traffic and decreases residence. Also, in Chapter Four I demonstrated that residence tends to decrease when patches contain fewer flowers, with a near-linear relationship between patch size and residence in matrix-style landscapes such as those I have used in this Sacrificial Shield study. Taken together, these results imply that if the foraging landscapes represented here were to actually contain additional smaller patches, the gene-flow predictions could potentially be underestimates. Consequently, the patterns of gene flow variation observed by manipulating spatial and reward availability factors of the Sacrificial Shield could still prove to be important.

The Sacrificial Shield implementation strategy appears to be an effective strategy for GM containment. The presence of a barrier of fields situated directly between two populations clearly decreases the level of bumble bee mediated gene flow between them. This occurs because the model bees tend to prefer moves to nearest neighbour resource sites, all else being equal. This effect is even more pronounced at the landscape-scale in which fields are separated by distances in the order of 1km or more. For a bee to consider moving to a field that is beyond a neighbourhood depth of one field, the perceived quality of the distant field would need to significantly exceed those of the nearest fields, which is unlikely to occur in most cases, especially when the landscape's reward provision is close to homogeneity.

Whilst the placement of the shield directly between the counties was clearly the best strategy for gene containment that I tested, even counter-intuitive placements of the shield at counter-intuitive yielded a slight reduction in gene flow levels. Since gene flow events can only occur

when a bee moves from a source field into a sink field, the mere presence of additional catchment fields decreases the probability of a move being made to a sink field. Bees will explore the shield fields as potential alternatives, and when the qualities of the fields across the landscape are identical, the likelihood of bees exploring these fields is effectively determined only by their location in their landscape.

Assuming that the shield is placed in its optimum position as a direct barrier between the counties, the level of gene flow from source to sink is predicted to be lowest when the shield is placed equidistant from both counties, and highest when the shield is close to either of the counties. This emerges because of the isolation effects I discussed in Chapter Four. As the shield is moved closer to the Conventional County, the county becomes less isolated, and the average residence within its fields decreases because there are nearby viable alternatives. We know from the E-Psi-b model (Cresswell *et al*, 2002) that a reduction in sink residence increases gene flow levels. When the shield is close to the GM County, residence in the Conventional County is also lower because the probability of moving from source to sink is highest in this instance. This may seem counter-intuitive at first glance, but it is important to reiterate that the probabilities of moves being made are derived from the proportions of moves made within the system during the course of the simulation. Levels of system wide inter-patch traffic are increased whether the shield is close to the GM County or the Conventional County, because the minimal cost of travel to and from the shield promotes movement. Consequently, the higher probability reflects an increased proportion of moves from source to sink, but not an increased number of absolute moves made from source to sink. As I discussed in Chapter Four, that is irrelevant for the purposes of gene flow calculation in which absolute numbers of visits are not considered.

When the fields that are closest to the borders of one county are closer to the fields of the other county, the level of gene flow is predicted to be higher. When the counties are closer together, inter-field distances for inter-county moves are lower, and therefore the average level of movement within the system is increased. Similarly, residences are shorter because there is a greater likelihood of a foraging bee having a nearer viable alternative site in which to forage. Whilst it has been shown that, in the case of bumble bees, simple isolation-based containment strategies do not eradicate gene transfer between populations (Ramsay *et al*, 2003), the results presented here suggest that the principle itself could at least reduce the levels of bee-mediated gene flow. Potentially, the efficacy of the 'increased distance' strategy is increased when coupled with other, more effective containment approaches.

A surprising result emerged from my exploration of shield reward availability manipulation. In general terms, we might expect that if the fields within the shield contained fewer nectar-bearing flowers then, all else being equal, bees would be less inclined to visit these fields, and there would be an increased likelihood of a bee moving from the source to the sink. However, the model predicts that the level of source-to-sink gene flow is lowest when the shield is lacking in resources. Whilst I did find that a poor quality shield increased the probability of a bee making a source-to-sink transition, this effect was far outweighed by a significant increase in residence in source and sink fields. This increase occurs because the poor quality of the shield fields all but removes them as viable foraging sites for the bees, who consequently stay longer in the other (superior quality) fields. A further catalyst for this change is that the counties are separated by a distance of 2km, making inter-county moves unattractive in most cases.

Clearly, the results I have presented here are preliminary, and should not of themselves constitute a containment strategy. However, the patterns that have emerged suggest the sort of landscapes that may promote gene containment, at least from the perspective of bee-mediated gene flow minimisation. Further investigation should focus on exploring these emergent containment patterns for landscapes that are more ecologically realistic, such as those containing complex matrices of varying size resource sites. This is important, because whilst I have identified a containment strategy, the economic cost of implementing this scheme would need to be assessed to determine whether the strategy is economically viable, as the costs could potentially outweigh the benefit of the lower economic value of GM-labelled produce (Belcher *et* al, 2005).

6.5 Summary

When applied to landscape-scale foraging scenarios, in which resource patches represent fields containing hundreds of thousands of flowers, and kilometre-scale distances separate fields, the model makes a number of predictions. First, the absolute level of movement between fields is predicted to be extremely low at such scales, because the costs incurred by moving between resource sites are significant. Since this translates directly into gene flow, the potential risk of bee-mediated gene escape for scenarios where GM containment is desirable is therefore predicted to be low. This is consistent with observed results (e.g. Damgarrd and Kjellson, 2005; Rieger *et al*, 2002), which invariably show low gene flow over long distances – indeed this is an 'iconic pattern' (Chapter Four).

However, real foraging landscapes contain a rich diversity of foraging sites, and bumble bees may visit intervening smaller patches en route to larger fields (Ricketts, 2001). The level of inter-patch traffic increases as the number of foraging sites available in the landscape is increased, which could in turn increase gene flow levels.

Second, it is predicted that there would be far more movement between patches when the resource availability is similar in all patches in the landscape. Bees tend to move more frequently in landscapes that are closer to homogeneity, because they are effectively searching for a superior foraging site that doesn't exist. In landscapes whose patches are more variable in terms of their reward yield, the average level of inter-patch traffic is predicted to be lower, because bees, being efficient learners, quickly learn the location of superior foraging sites, and generally remain loyal to such patches. From a gene containment perspective, variability in standing crop levels across a landscape would therefore seem to be a potential means of reducing genetic transfer across the system. Variability in resource levels among fields is likely caused by difference in crop varieties (Pierre *et al*, 1999) or by variation in soil moisture (Chapter Five; Wyatt *et al*, 1992; Carroll *et al*, 2001). However, minimization of such variability does not seem to be a viable containment strategy.

Third, the implementation of a direct barrier, or shield, of 'sacrificed' fields between two populations can significantly reduce the potential for gene flow from one population to another, because bees generally prefer to switch to nearer resource sites. To ensure gene flow minimisation is upheld, the populations should be separated as much as possible, and the shield should contain fields that offer very little nectar and situated equidistant from both populations.

In the final chapter, I present a discussion of the project as a whole and its limitations, and suggest potential avenues of future investigation to extend the findings from this research study.

Chapter Seven : Implications and Further Work

7.1 Introduction

Initially, I will review the insights and generalizations achieved by the analysis of the behaviour of economiocally motivated RL foragers in HARVEST. I will then critically evaluate shortcomings and suggest direction for further work.

7.2 Implications

7.2.1 Gene flow levels in landscapes

Previous studies have found that levels of bee-mediated gene flow are generally low (e.g. Rieger et al. 2001, Cresswell 2006), which has been attributed in part to pollinator behaviour (Hoyle *et al*, 2007; Cresswell *et al*, 2002; Walklate *et al*, 2004). Using the HARVEST model of pollinator behavior, I have succeeded in generalising this finding. HARVEST predicts that bees will be responsible for only minimal levels of genetic movement between populations in km-scale landscapes, which are realistic foraging scales for assessing bee-mediated gene flow (Osborne *et al*, 1999; Westphal *et al*, 2006). Primarily, this emerges because bees typically move infrequently between foraging sites at the landscape-scale, as the distances between sites are economically prohibitive. Consequently, the probability that a bee directly transfers pollen from a source population to a sink population is low, and when it does occur, the influence of the extrinsic pollen is overwhelmed by the spread of intrinsic pollen, because bees visit many flowers in large patches such as agricultural fields (Cresswell *et al*, 2002).

I have shown that this low-level of bee-mediated gene flow is a generalised finding, because regardless of how the foraging landscape is configured – either spatially or in terms of resource availability – the level of gene flow from source to sink alters little, and certainly remains far below EU tolerance threshold levels for non-GM crop purity (CEC, 2003). The modelling approach I have employed here, HARVEST, has generalised beyond the limited number of empirically tested scenarios to predict that the observed low levels of bee-mediated gene flow are highly likely to be a universal pattern.

It is possible that smaller-scale patches could be vulnerable to breaches of GM containment (Cresswell *et al*, 2002), as HARVEST predicts that smaller patches reduce the number of

flowers that bees visit, and increase levels of inter-patch traffic in the system, because bees are more likely to encounter flowers that have recently been emptied, either by themselves or their competitors. Whilst smaller-scale landscapes are not potentially problematic in themselves in terms of gene containment, because they can be manipulated easily and genes are exchanged in the same adaptive zone, any smaller-scale landscape can be thought of as also being a part of a larger scale foraging landscape. Indeed, real-world foraging landscapes are typically comprised of both large and small-scale foraging sites (Ricketts, 2001). For the case of bumble bees, it is likely that they will consider a range of foraging sites within their large foraging radius. We also know from HARVEST that changes in landscape-size can affect levels of patch residence and traffic (Chapter Six), and that variability in patch size can affect these levels as well (Chapter Four), which has also been demonstrated empirically (Klinger *et al*, 1992; Goodell *et al*, 1997). Taken together, the implication is that gene flow levels could be higher than HARVEST has predicted for the landscapes that were tested. However, the increase is not inevitable because both empirical studies (Pleasants, 1981; Ott *et al*, 1985; Cartar, 1991; Dall *et al*, 2005) and HARVEST trials (Chapter Three; Chapter Five) have found that bees are very efficient learners, and may increasingly reject smaller-scale patches because of the increased encounters with depletion effects. Clearly, further investigation is needed.

7.2.2 Containment strategies

HARVEST predicts that gene flow could be near-eliminated by implementing a 'sacrificial shield' – a barrier of crops whose genetic purity is willing to be sacrificed to preserve the purity of other populations. This finding is in line with previous investigations that have found that the use of such barrier crops is an effective gene containment strategy (Morris *et al*, 1994; Reboud, 2003). Given the very low levels of bee-mediated gene flow that HARVEST predicts however, and assuming that the more realistic landscape simulations described above do not significantly increase these predicted levels, the question should be raised as to whether such containment strategies are worth implementing. From a purely economic perspective, GM crops offer the advantage of lower production costs, because there is a diminished need for the use of pesticides, and GM crop production is precise and efficient (Raney, 2006). Potentially, the implementation of barrier crop containment strategies could negate this advantage and prove to be economically unfavourable, because potentially large areas of crop would need to be set aside to be sacrificed and later destroyed to stop the spread of GM genes (Morris *et al*, 1994).

One scenario in which the costs of containment strategies would be worthwhile is when GM crops are utilised for molecular pharming, in which plants are used to produce pharmaceuticals.

Consequently, the tolerance for the presence of genes in food plants that originate from plants used in molecular pharming is low, both from a regulatory and a public acceptance perspective (Horn *et al*, 2004), primarily because of safety concerns. Molecular pharming is a potentially lucrative industry, and the adoption of containment strategies that severely minimise gene flow from such populations into the food chain could not only overcome regulatory obstacles, but also increase public support for this production method, thereby potentially increasing growth of the industry further.

7.2.3 Iconic patterns

HARVEST generalised the iconic patterns of bee-mediated gene flow that have been shown to emerge when population sizes and the distances between populations are varied (Cresswell, 2006). Specifically, HARVEST predicts that the level of gene flow between a source and a sink population diminishes as the distance between the populations is increased (Bateman, 1947; Damgaard and Kjellson, 2005; Fenster, 1991), and as the size of the sink population is increased (Klinger *et al*, 1992; Goodell *et al*, 1997). I also demonstrated how quantitative accuracy could potentially be achieved by manipulating replenishment frequency parameterisation.

However, HARVEST did identify a counter-intuitive 'quirk' in which the iconic pattern relating distance to gene flow was not upheld, and the level of gene flow actually increased as the distance between source and sink populations was increased. Specifically, this occurs when an increasing distance between the source and sink populations implies a decreasing distance between the sink and a third 'catchment' population. In such a scenario, gene flow levels are highest when the distance between the source and sink populations is greatest, because of the interference of the catchment site, which affects residence levels. This is an interesting result, and implies that the gene-flow distance iconic pattern is not universal. However, it is unlikely that this scenario is ecologically representative, because real-world foraging landscapes contain arrays of patches in which increasing distance from one patch can imply increasing proximity to *numerous* other patches (Ricketts, 2001). Indeed, preliminary investigation with matrix-style landscape configurations (Chapter Four) showed that the iconic pattern was replicated for such scenarios, suggesting that realistic landscapes will uphold the iconic pattern. Nonetheless, the finding does bring to light the sensitivity of bee-mediated gene flow relationships to small-scale landscape configuration, highlighting the importance of modelling-based approaches to query alleged universal patterns.

7.2.4 Efficient foraging despite incomplete information

The rules for HARVEST were derived simply from a set of principles based on expectations of how bees should behave, given that they are economically motivated foragers who are food-focused (Best and Bierzychudek, 1982) and forage for the evolutionary success of the colony (Heinrich, 1979). Also, a fundamental requirement of the model was that model bees should be incompletely informed foragers – that is, be unaware of the actual distribution of food availability within a foraging environment – in order to circumvent the justifiable criticisms of unrealistic traditional Optimal Foraging Theory models (Pyke, 1984), and to ensure that the powerful learning characteristic of bumble bees was captured (Dall *et al*, 2005). In spite of this, and being provided with only simple decision making mechanisms to ensure biologically plausibility, HARVEST bees prove to be extremely efficient learners when presented with the patch choice problem in uncertain landscapes (Chapter Three).

The performance of HARVEST bees leads to two implications. First, it suggests that the Reinforcement Learning mechanism is a powerful model for learning agents faced with problems to which a direct solution is unknown. This is somewhat unsurprising, as the RL approach has previously been shown to be very effective when applied to the ecological domain (Niv *et al*, 2002). Additionally, the linear operator learning rule used for updating patch quality estimates has been shown before to be a preferred learning rule for learning-based foraging theory (Beauchamp, 2000). Second, the near equivalence of the foraging performance obtained by learning bees and their omniscient counterparts suggests that the learning bees in the model are extremely efficient at locating preferential patches of resource, and rejecting those that lead to inefficient foraging. This agrees with previous findings that have shown that bees are efficient adaptors to new information (Pleasants, 1981; Ott *et al*, 1985; Cartar, 1991), and lends further credence to the model as a simulator of bumble bee foraging behaviour.

7.2.5 Foraging behaviour at the small scale

Fundamentally, HARVEST demonstrates that the frequent inter-patch flights of bumble bees at the small-patch scale is strategic, because bees are risk-averse (Harder and Real, 1987; Cartar, 1991; Cartar and Abrahams, 1996) and are therefore trying to avoid depletion effects that decrease foraging efficiency. HARVEST mimics this strategy mechanistically, because bees in the model are sensitive to 'runs of bad luck', and implicitly contextualise these encounters in terms of the perceived quality of alternative foraging sites.

7.2.6 Critical evaluation of the model's predictions

Given that HARVEST predicts only negligible levels of bee-mediated gene flow between populations at the landscape-scale, it is prudent to question the validity of the model's predictions. After all, there is currently no way to empirically test the model's predictions at the larger landscape-scale, hence the need for the model, and at the smaller-scale I found quantitative discrepancies between the model's predictions and empirical data. I now discuss these discrepancies in the context of their ability to diminish confidence in the model.

HARVEST typically over-estimates both the number of flowers that bees will visit in patches, and the frequency with which they will move between patches. Whilst HARVEST achieves qualitative accuracy in comparisons between landscape scenarios, accuracy in both residence and traffic prediction is required to formulate reliable gene flow estimates. It is possible that the quantitative inaccuracies arose because the replenishment frequencies that I simulated were incorrect. We know that the frequency with which plants replenish nectar supplies can influence patch departure mechanisms (Chapter Four; Chapter Five), because the likelihood of encountering a flower bearing no nectar is changed. Consequently, gene flow levels can be affected by floral replenishment rates. I had no direct measurement of replenishment rate in the empirical study with which HARVEST's predictions were compared, and so it is possible that the replenishment rates being simulated were inaccurate.

If an inaccurate simulation of replenishment frequency were indeed the primary reason for quantitative inaccuracy in the small-scale predictions of HARVEST, the validity of the model should not be affected at the landscape-scale for two reasons. First, as I discussed in Chapter Six, the large-scale foraging sites that dominate landscape-scale foraging by bumble bees are so large that bees are unlikely to cause a net decrease in standing crop levels. Therefore, the simulation of depletion and replenishment at the landscape-scale is unnecessary, and its potential accuracy at the smaller-scale irrelevant. Second, the model achieved qualitative accuracy in terms of its ability to replicate commonly observed iconic patterns of bee-mediated gene flow between populations (Chapter Four). This implies that the mechanistic basis of the model is valid, but may produce inaccuracies as the result of poor parameterisation which, as I have just discussed, is less significant at the landscape-scale.

Quantitative inaccuracies may have also emerged because HARVEST simulates a 'closed system', whereas real bees are free to fly to and from the set of patches being investigated.

Indeed, bees in the six patch experiment had other foraging sites nearby – such as patches of wildflower - that they visited. HARVEST bees are constrained to the set of foraging sites under investigation, but are equipped with nectar carrying capacities that approximate those of real bees (Cresswell *et al*, 2000; Cresswell *et al*, 2002; Chapter Four), and are allowed to forage for realistic periods of time (Osborne *et al*, 1999; Hodges, 1985a; Plowright and Galen, 1985). Therefore, HARVEST bees may have spent longer in patches because they had fewer alternative locations to choose from than their real-world counterparts. Ironically, in a closed system, this would also increase inter-patch traffic levels, because all moves made would be moves between the patches in the system. This would imply over-estimates in both residence and traffic levels, which is what I found.

The closed nature of the system could also be a problem when simulating at the landscape-scale, and so may have introduced inaccuracies in my landscape-scale predictions. However, this again becomes a question of parameterisation rather than of the accuracy of the model's behavioural mechanisms, because the closed system limitation could be overcome simply by including all of the alternative patches available to a bee within their real-world landscape. Given the large foraging radius of bumble bees, however, a more sensible compromise would be to sufficiently increase the resolution of the landscape map being simulated.

In bumble bee colonies, newly hatched workers emerge contiuously throughout the season (Goulson, 2003). The presence of naive bees in the empirical study may have created noise in the empirical data, because if a proportion of bees that visited the field site had not visited the grid before, the levels of residence and inter-patch traffic obtained may be misleading, because such bees would need to learn the qualities and configuration of the landscape presented to them. It may therefore be useful for any future replication of the experiment to mark visiting bees and record the number of bees that are repeat visitors. HARVEST already has the capacity to simulate a specific mixture of naive and experienced foragers; the parameter Ω, with $0 \leq \Omega \leq 1$, defines the probability that a bee that has returned to the nest will become a naive forager on their next foraging bout, by having all of their estimated patch qualities and action-values reset to default values. Ω therefore approximately emulates a proportion of naive foragers in a system.

It would therefore seem that the most likely explanations for inaccuracy in HARVEST's predictions at the small-patch scale are either rendered irrelevant at the landscape-scale, or can be easily resolved by improved parameterisation. Qualitative agreement across the study

suggests that the model itself is a good replicator of bumble bee behaviour, even if the mechanism at its core is not an accurate representation of how real bees' minds operate, which was not the goal of this psychologically-neutral investigation. The extrapolation of the model to the landscape-scale scenario is valid because the model was designed from first principles of bee behaviour and economic optimisation (Chapter Two), which are likely to be scale-independent, and optimised for larger scale foraging in which sub-optimal decisions are more costly.

7.3 Suggestions for further Work

7.3.1 Incorporation of systematic foraging and traplining

HARVEST bees select flowers at random from the pool of flowers available in a patch. This is useful for an initial simulation of bee activity, because it allows us to emulate the uncertainty that a real bee would experience when visiting a new flower, and it eschews the need to accurately capture within-patch spatial movement dynamics. Nonetheless, real bumble bees have been shown to be systematic foragers (Thomson, 1996; Ohashi *et* al, 2008; Williams and Thomson, 1998). A HARVEST bee can potentially select to visit a flower that they have just visited, which would be unrealistic because it is likely that a bee would realise that the nectar in that flower would not have had a chance to replenish since the last visit.

The model could be extended to incorporate systematic foraging behaviours. A simple extension could limit the possible flowers that a bee could visit in a patch, based on the time since the bee last visited the flower. A duration threshold could be defined, such that flowers that have been visited more recently are unavailable as valid flower choices for the bee. Alternatively, a more sophisticated mechanism could be implemented, in which the probability that a bee selects a flower is weighted according to the time since the flower was last visited by the bee. Any such implementation would require both empirical justification and validation; the mechanism employed would need to reasonably approximate real-world systematic foraging dynamics, and the model's predictions would need to be re-assessed to ensure qualitative and quantitative accuracy was maintained.

Systematic foraging also extends to patch choice mechanisms, in which traplines of repeated visit sequences may be followed by foraging animals. In my prior empirical study conducted in 2007, bee behaviour was analysed in matrices of individual plants, separated by metre-scale distances. As part of the study, the visitation sequences of visiting bees were recorded. Using

the Gestalt approach of pattern recognition via sequence matching algorithms in Python, the data was analysed to look for closely matching sequences of visits, which would indicate the presence of traplining behaviour. The results showed that the bees visiting the experimental site did not follow traplines within the matrix. Furthermore, identical sequence matching algorithms were applied to visit sequence data extracted from the model. I found that the bees in HARVEST also did not follow traplines through their foraging landscapes. It is therefore initially tempting to conclude that real bees do not generally trapline, and that the model supports this. This is likely an erroneous simplification, however.

There are a number of reasons why bees in the 2007 study may not have followed traplines through the matrix. The configuration of the matrix may have been too dense – plants were arranged in a grid with a separation distance of just 0.9m between cardinal neighbours. Consequently, bees exploring central areas of the matrix may have become disoriented, as all directions would have looked similar. It is thought that bees use both landmarks (Goulson and Stout, 2001; Plowright and Galen, 1985; Birmingham and Winston, 2004; Colborn *et al*, 1999; Wehner *et al*, 1996) and site boundaries (Plowright and Galen, 1985) to orient themselves. Bees would need to be able to orient themselves within their foraging landscape for traplining to emerge.

It is also possible that the conduct of the experiment influenced the lack of traplining behaviour. Throughout the course of the experiment, which took place over six days, the grid was increased in size. Each day, new plants were added to the edges of the matrix as they came into peak bloom, expanding the matrix outwards. Whilst the rationale behind this was to promote homogeneity of standing crop levels across the grid, the daily change in configuration may have caused bees to re-learn the matrix afresh each day. Of course, this explanation necessitates the assumption that some of the bees were returning visitors from the previous day, which would need to be validated using bee marking methods (Saville *et al*, 1997). Regardless of the actual influence on traplining of changing the experimental configuration throughout the study, I revised this approach for the 2008 study to minimise the potential for disruption from such variation.

Even though traplining was not observed in my study, a significant base of literature presents evidence of such behaviour exhibited by bumble bees (e.g. Thomson, 1996; Ohashi *et* al, 2008; Williams and Thomson, 1998). It is therefore appropriate to consider how the model bees in HARVEST may be enticed to trapline. Essentially, a trapline would emerge because an animal

is aware in some way that a previously visited resource site will have replenished supplies at some point in the future (Williams and Thomson, 1998). In the case of bumble bees, this would imply that bees are aware that flowers have renewing supplies of nectar, and therefore patches that were previously considered to offer poor quality foraging conditions may have improved. Bees in HARVEST only ever update their perception of the quality of the patch in which they are currently foraging. The estimated qualities of all alternative patches remain static. A simple extension of the model may involve a 'floating estimate' scheme in which, after every unit of time that elapses in the simulation, a bee not only recalculates its estimate of the current patch, but also increases its estimate of all other patches by a defined amount. This amount may be a constant, or may be derived from a function of nectar secretion over time. The 'floating estimate' scheme would introduce a second trigger for an inter-patch move – when the action-value of a 'move' action has increased beyond that of the 'stay' action. Logically, such an implementation should encourage bees to move to patches that they visited less recently, all else being equal, potentially emulating a traplining mechanism and, crucially, doing so using an ecologically justifiable mechanism. The concept of improving non-current patch estimates has also been used elsewhere, and has been shown to improve foraging efficiency (Groβ *et al*, 2008).

7.3.2 Incorporation of continuous nectar distributions

Currently, HARVEST implements a binary reward scheme, in which flowers are either full with nectar, or contain no nectar at all. Real floral nectar distributions are continuous because plants gradually replenish nectar levels in flowers over time (Pierre *et al*, 1999; Koltowski, 2002). Therefore, to accurately simulate real-world rates of replenishment, HARVEST should be altered to allow for continuous quantities of nectar in flowers. Rather than flowers being either 'full' or 'empty', flowers could be defined by the precise quantity of nectar they hold. Patch qualities (and their estimates) could be expressed as the mean level of nectar per flower in the patch. This change would also have consequences for the updating of patch quality estimates, but the Linear Operator learning rule used in the model (equation 2.1) is already flexible enough to accommodate this; the value *s(i)* represents the quantity of nectar obtained from the flower sample. In general terms, the change to a continuous nectar distribution model should, theoretically, improve the accuracy of bee learning, and could increase quantitative agreement with empirically-obtained data.

7.3.3 Specifying spatially-explicit flowers within patches

HARVEST is spatially explicit in the sense that its foraging landscape is comprised of patches that are explicit locations within the landscape, separated by tangible distances. However, patches in HARVEST are not spatially-explicit themselves, and represent only points on the foraging map that occupy no space. It has been shown that the level of gene flow between a plant in a source population and one in a sink population can vary according to their spatial locations within their respective patches (Klein *et al*, 2006), and therefore gene-flow levels between populations can depend upon the size and shape of the populations.

HARVEST already abstractly captures the concept of patch size, but the bees have no awareness of the size of the patches in which they forage, and the size of the patch has no direct influence on a patch departure decision (though smaller patches do increase the probability of a bee selecting a recently emptied flower and, therefore, the probability of an inter-patch move being instigated). Real bees, being spatially aware animals (Chittka *et al*, 1999; Wehner *et al*, 1996; Pyke and Cartar, 1992; Goulson and Stout, 2001; Manning, 1956), are likely to be conscious of differences in patch size. These differences in themselves may influence patch departure decisions. For example, a bee foraging in a very small patch may leave the patch more quickly because it realises that there are not enough flowers for them to be sufficiently satiated, or because it realises that a smaller number of flower encounters is a more representative sample in smaller patches. Such awareness could be built into the model by either defining triggered departure thresholds that consider both remaining capacity and patch size or, preferably, by weighting the influence of an individual flower sample according to the size of the patch. The latter alternative could be implemented by changing β to a dynamic variable that can change over the course of the simulation.

In order to simulate patches of different shapes, however, within-patch simulation could be made spatially explicit. Spatially-explicit patch representations could be represented using the current model framework, by using patches to represent individual plants, and spatially grouping these plants into distinct patches. Alternatively, the model could be extended to accommodate an additional layer of spatial simulation beyond the patch layer. Either way, the model could then accommodate patches that vary in terms of their shape. This may also be an important consideration if within-patch systematic foraging was to be simulated.

7.4 General Summary

The landscape-scale movement of bumble bees has been a mystery, as empirical approaches have largely lacked feasibility (Osborne *et al*, 1999; Dramstad, 1996), and traditional modelling approaches have typically ignored or downplayed the relevance of learning mechanisms (Emlen, 1966; Pyke, 1984, Rodriguez-Gironés and Vásquez, 1997; Kohlmann and Risenhoover, 1998; Keasar *et al*, 2002) and a forager's changing internal state (Mangel and Clark, 1986), which are arguably vital when modelling strong learners such as bumble bees (Keasar *et al*, 2002; Dukas and Real, 1993) that are also gradually accumulating a payload. HARVEST builds upon the core principles of Optimal Foraging Theory (Emlen, 1966; MacArthur and Pianka, 1966), Individual-Based Modelling (Grimm and Railsback, 2005) and Reinforcement Learning (Sutton and Barto, 1999), defining a model that represents bumble bees as explicitly individual economically-motivated learning agents.

My initial validation of HARVEST's predictive capabilities is encouraging. It is able to qualitatively replicate iconic bee-mediated patterns of gene flow, and with more realistic parameterisations could potentially achieve quantitative accuracy. Its quantitative predictions are weakest at the scale of the small array of flower patches, but its shortcomings can be explained, and with further tuning could potentially be eliminated.

Sufficient assessments of the risks to crop purity are a vital ingredient when introducing GM crops into existing agricultural systems (Conner *et al*, 2003). The risk from bumble bees of GM gene escape has previously been highlighted (Ramsay *et al*, 2003) and downplayed (Hoyle *et al*, 2007), and HARVEST's landscape-scale predictions of gene flow tend to support this notion of only trivial influence from bumble bees. Nonetheless, it would be misguided to rule out the risk from this pollen vector based only on preliminary results. Therefore, HARVEST should be considered as an appropriate preliminary platform, whose foundations can be built upon. Furthermore, the HARVEST approach could be used to enhance understanding of bumble bee dynamics, both in respect of the problem outlined in this study, as well as providing a tool to assist in conservation strategies for plants and the bees themselves, by characterising the resource richness of landscapes. However, the study of learning is fundamental in behavioural ecology and psychology (Pyke, 1984; Hirvonen *et al*, 1999; Keasar *et al*, 2002; Olsson & Brown 2006; Chapter One), and therefore an approach based on RL agents could be extended beyond the domain of bumble bee simulation, and could potentially guide a renewed development of Optimal Foraging Theory, with ecologically-derived learning mechanisms constituting the impetus and not the impediment.

Appendix A : Transition Matrix Extraction – Worked Example

HARVEST outputs the transition matrix – the proportions of specific inter-patch moves made by the colony – as a comma separated values (CSV) file. The matrix allows us to extract the proportion of moves made from a given starting patch to a given destination patch. In the file, data for each trial in the trial set is presented as a separate matrix, sequentially. Within each matrix, each row (or new line in the file) indicates a separate starting patch. The row number (starting from row 0) indicates the identifier number of the starting patch. Each column (or comma separated entry on a line) indicates a separate destination patch. The column number (starting from column 0) indicates the identifier number of the destination patch.

Patches in HARVEST are numbered sequentially, starting from patch 0. Patch 0 always represents the nest. Moves to and from the nest are not relevant for gene flow calculation, because the nest is not a patch of resources. Therefore, HARVEST outputs moves to and from the nest as the total number of absolute moves, rather than a proportion of moves. These moves are not included in the calculation of total moves made within the foraging system. All other patch-to-patch movements are expressed as proportions, where the entry at (row *a*, column *b*) represents the proportion of actions taken by all bees in the colony that were moves from patch *a* to patch *b*. If $a = b$, then the entry represents the proportion of actions that were 'stay' actions in patch *a*.

Therefore, in general terms, a transition matrix file for a trial set of 2 trials with 2 (non-nest) patches has the following structure structure (where Px = Patch x, N = Nest):

No. of stays in N, No. of moves from N to P1, No. of moves from N to P2

No. of moves from P1 to N, Proportion of moves from P1 to P1, Proportion of moves from P1 to P2

No. of moves from P2 to N, Proportion of moves from P2 to P1, Proportion of moves from P2 to P2

Here is an example output file from a real HARVEST trial set containing two trials and 6 (non-nest) patches :

0,21,42,63,21,42,23,

21,0.170206845376,0.00301456265653,0.00292180688248,0.00452184398479,0.00458213523792,0.00222613857713,

42,0.00361747518783,0.134124849272,0.00422038771914,0.00422966329654,0.00211019385957,0.00317224747241,

42,0.00459141081532,0.00361747518783,0.160736480846,0.00222613857713,0.00105741582414,0.00542621278175,

21,0.00361747518783,0.00317224747241,0.00400704943883,0.105741582414,0.00482330025044,0.00264353956034,

21,0.00176235970689,0.00331601892218,0.00352471941378,0.00542621278175,0.190334848344,0.00301456265653,

21,0.00445227715425,0.00422966329654,0.00271310639087,0.00125220294963,0.00422038771914,0.135145162786,

0,0,63,63,42,21,23,

0,0.174473610982,0.00211019385957,0.00292180688248,0.00753640664131,0.00211483164827,0.00222613857713,

21,0.00422038771914,0.125034783415,0.00301456265653,0.00105741582414,0.00391893145348,0.00352471941378,

63,0.00584361376496,0.00512475651609,0.173569242185,0.0017809108617,0.00176235970689,0.00331601892218,

42,0.00301456265653,0.00281977553103,0.00267136629255,0.115109915592,0.00422038771914,0.00417400983211,

21,0.00176235970689,0.00331601892218,0.00387719135516,0.00301456265653,0.184120211483,0.00482330025044,

21,0.00311659400798,0.00176235970689,0.00542621278175,0.00250440589927,0.00452184398479,0.126194230591,

Here are some example statements that could be made about the results :

(a) In the first trial, 0.2% of all actions made by the colony were moves from patch 2 to patch 5
(b) In the first trial, 19% of all actions made by the colony were decisions to stay in patch 5
(c) In the second trial, bees moved directly from the nest to patch 2 63 times

For the purposes of calculating gene flow using the *E-Psi-b* model, we need to populate *E* using the transition matrix. In the example above, if the *source* patch were patch 3, and the *sink* patch were patch 4, we would set *E* to be 0.002, because this is the mean of the total transitions from patch 3 to patch 4 in trial 1 (0.00222613857713) and trial 2 (0.0017809108617).

Appendix B : Redundancy of the Non-Greedy Exploration Parameter

In machine learning, it is common for a parameter to be included that determines the probability that a learning agent makes a sub-optimal decision, for the purpose of exploring alternatives to see if they are superior (Sutton and Barto, 1999). HARVEST includes such a parameter, ε, where $0 \leq \varepsilon \leq 1$. ε represents the probability with which a bee will choose to be *non-greedy* for the next choice of action. If a bee is being *greedy*, it will choose the action that has the lowest corresponding action-value, breaking ties randomly. If a bee is being *non-greedy*, it will choose randomly from all available actions, regardless of action-value, including any actions that would be chosen if the bee were being *greedy*. An assessment is made for each bee before every action they take, to determine whether or not the decision will be made greedily.

In one of the earlier experiments that I conducted with HARVEST, I found that ε was an unnecessary inclusion, given that the model uses a linear operator learning rule that incorporates the β learning rate parameter. I tested a variety of landscape configurations, with values of ε and β independently varied, and ranging from 0.0 to 1.0. I here present the results from one example experiment from this set of trials, but all experiments yielded the same pattern of results. The results I present here are arguably the most useful, because they are taken from a highly variable landscape configuration, which should theoretically be an optimal landscape for the inclusion of ε.

Methods

I configured a landscape of three fields, each of which contained 10,000 flowers in total. Fields were labelled Field A, B and C. Field A was the designated poor quality field, and contained just 1,000 full flowers ($Q(A)$ = 0.1). Field C was the designated high quality field, and all flowers were full ($Q(C)$ = 1.0). Field B was a moderate quality field, containing 4,500 full flowers ($Q(B)$ = 0.45). Fields were separated by a distance of 10 time units (approximately 210m), and all inter-field (and nest to field) distances were equidistant.

Depletion and replenishment effects were enabled, despite the landscape-scale parameterisation, to generate more variability by using a dynamic landscape. The replenishment model differed slightly in this early version of HARVEST, because flowers were not explicitly modelled. Each

field was set to increase the number of full flowers by 1 for every 2 time units that elapsed. A time unit represented 3 seconds of real world time.

I tested β values of 0.0, 0.1, 0.2, 0.4, 0.6, 0.8 and 1.0, and ε values of 0.00, 0.01, 0.1, 0.2, 0.5 and 1.0, with all possible combinations of the two parameters. 10 bees were simulated in each trial, and bees were allowed to forage for 100 foraging bouts per trial. 10 trials were simulated per trial set.

Results

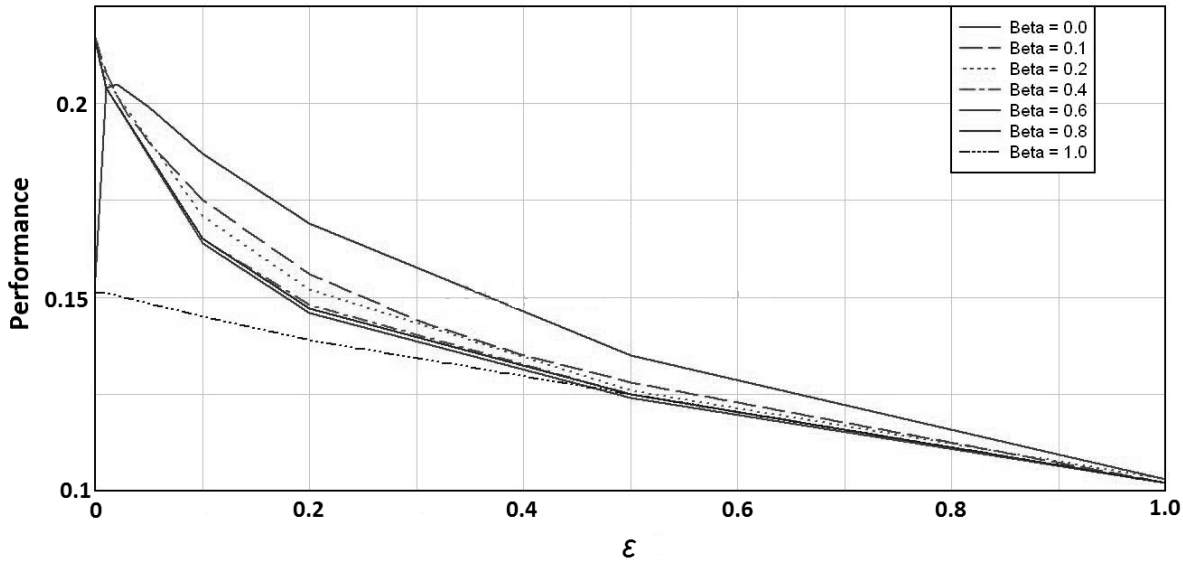

Figure B1. Mean foraging performance (y-axis) per bee with varying values of ε (x-axis) and β. Averages taken across trials.

Foraging performance is highest when $\varepsilon = 0$ – when bees always behave greedily – for all β values except 0 (figure A1). When both ε and β are 0, foraging shows the same lack as efficiency as when bees are most sensitive to individual flower encounters (when $\beta = 1$). Foraging performance generally degrades with increasing probability of being non-greedy (ε) and, to a lesser extent, as flower encounter sensitivity (β) increases.

Discussion

The results of the experiment show that, even in a variable environment, choosing an action that is considered sub-optimal is never beneficial in terms of foraging efficiency. The HARVEST model does not require the ε parameter, because it incorporates the β learning rate parameter, which implicitly determines a probability of exploration by defining the weight given to an individual flower encounter. HARVEST bees will therefore remain loyal to their patch when the foraging efficiency they are experiencing therein is 'good enough' (that is, higher than the estimate of an alternative), and will explore other patches when they encounter stochastic 'runs of bad luck' in their current patch. ε is redundant in the HARVEST model, because bees are already exploring patches sufficiently. By imposing enforced exploration moves on bees, the efficiency with which bees forage is degraded, because bees are very good at identifying 'better than average' patches using the β-based learning algorithm (Chapter Three), but are forced to move away from these patches by ε.

Appendix C : HARVEST Simulation Screenshots

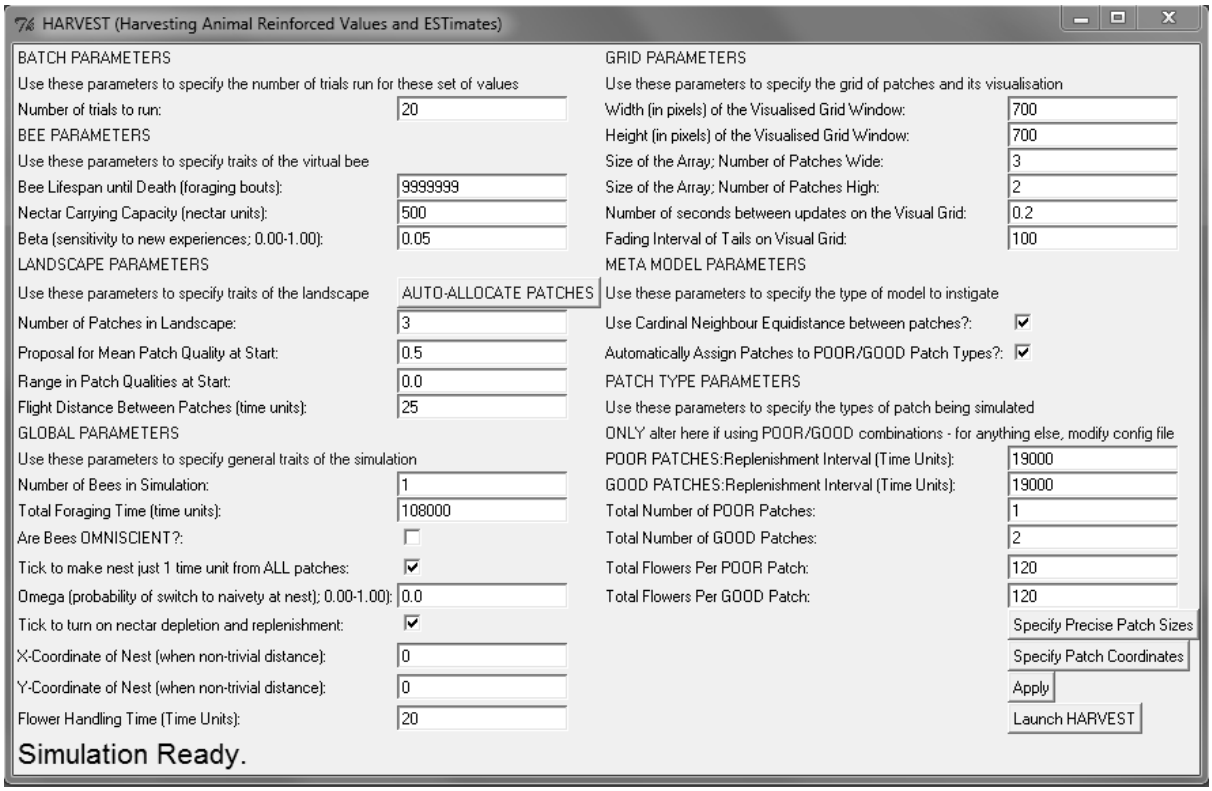

Figure C1. Main configuration window of HARVEST. Allows specification of parameter values before simulation is conducted. 'Launch HARVEST' button saves the parameter values, and then starts the simulation with the specified parameter values, whilst 'Apply' button saves changes without launching the simulation.

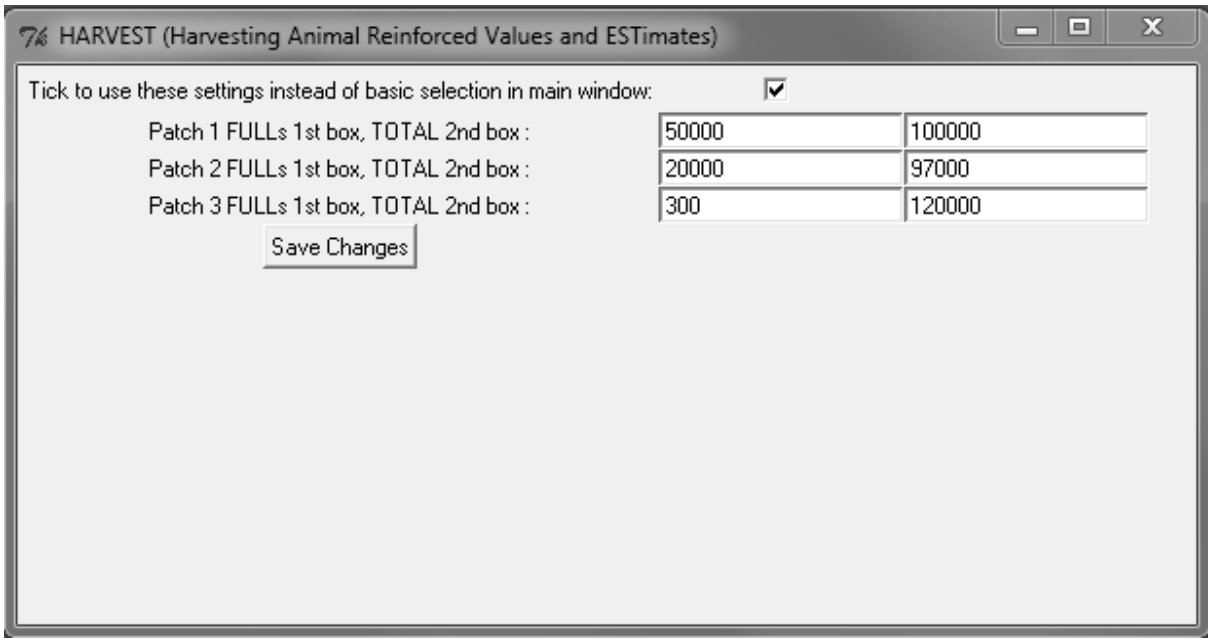

Figure C2. Window that provides ability to specify precise patch sizes for each patch in the landscape. Accessed via 'Specify Precise Patch Sizes' button in main configuration window.

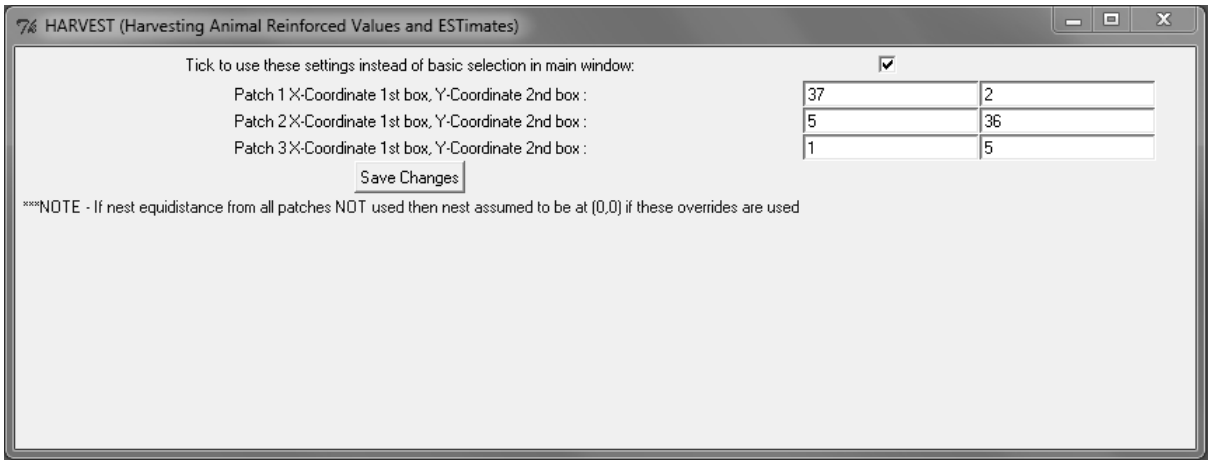

Figure C3. Window that provides ability to specify exact coordinates for patches within the landscape, rather than using the default matrix-style configuration. Accessed via 'Specify Patch Coordinates' button in main configuration window.

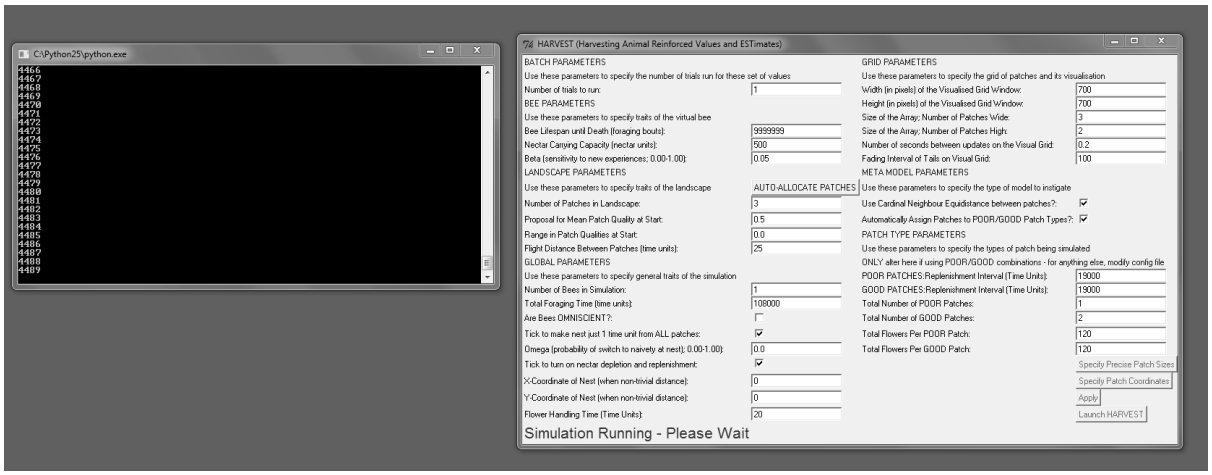

Figure C4. Screen capture of HARVEST when the simulation is running. Command prompt window outputs time on the global clock after every time unit. Main configuration window warns that the simulation is running, and disables buttons to avoid changes being made to parameter values in the middle of the simulation.

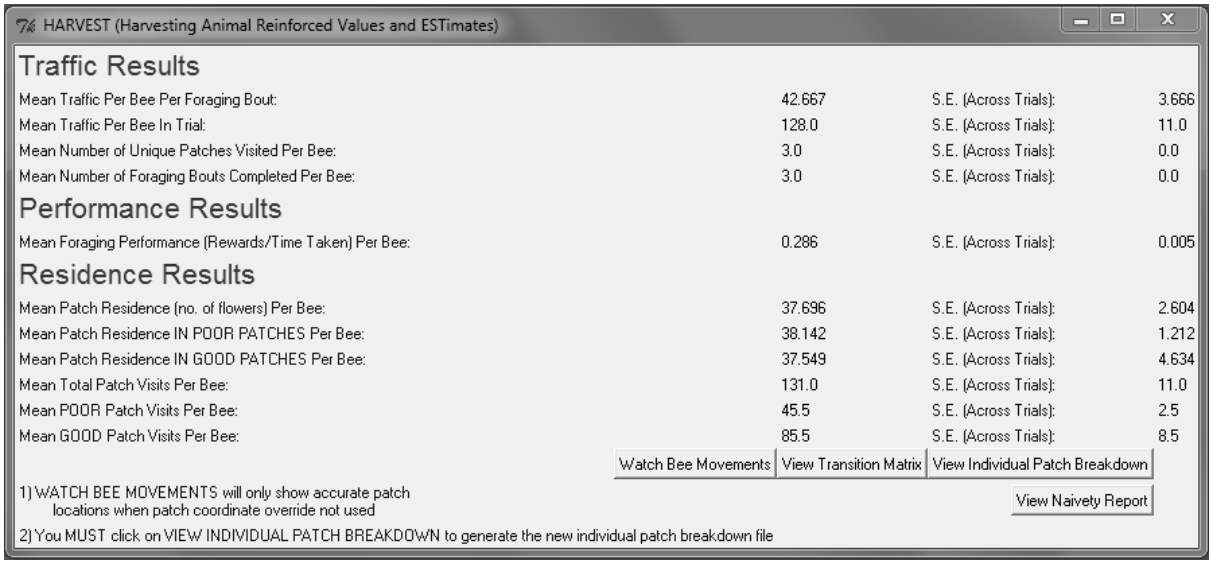

Figure C5. Main results window. Displays traffic, residence and foraging performance results, with averages taken across the trials that had been run in the batch, and S.E. indicating ± one standard error.

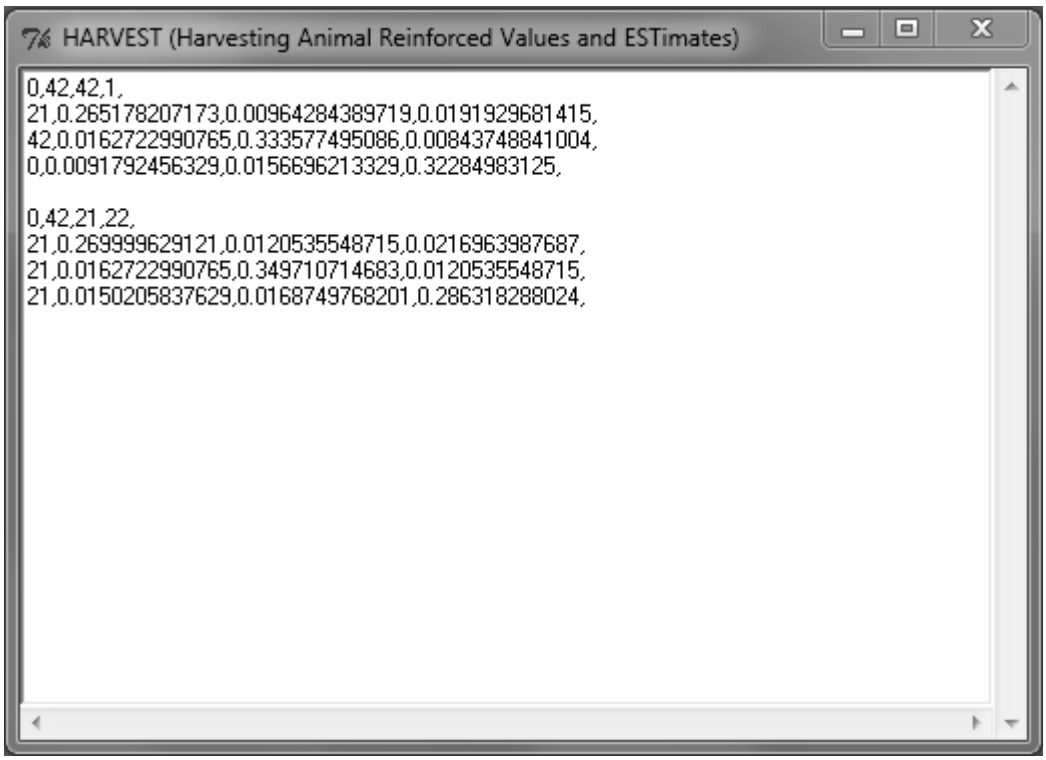

Figure C6. Window allowing read-only access to the transition matrix file (see Appendix A). Accessed via 'View Transition Matrix' button in main results window.

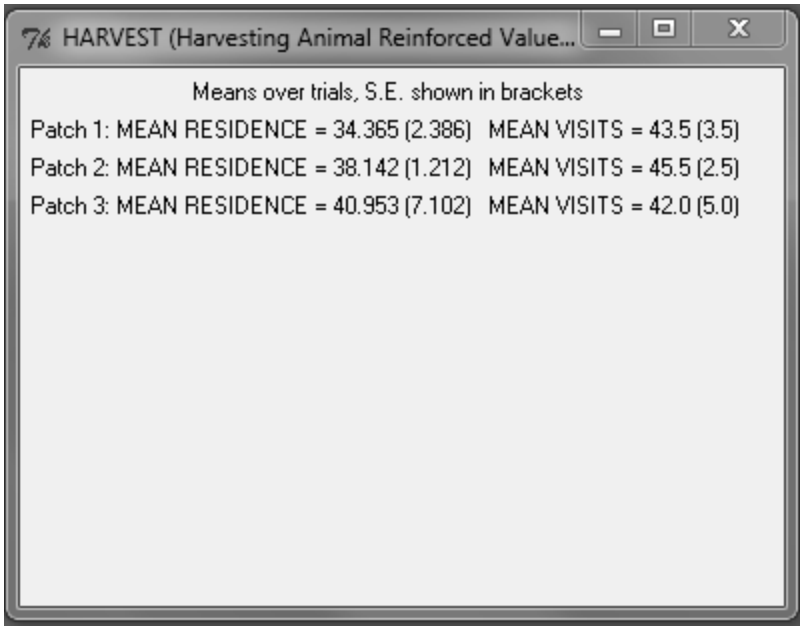

Figure C7. Window showing breakdown of residence and visit numbers to each individual patch in the landscape. Averages taken across trials, with ± one standard error (among trials) shown in parentheses. Accessed via 'View Individual Patch Breakdown' button in main results window.

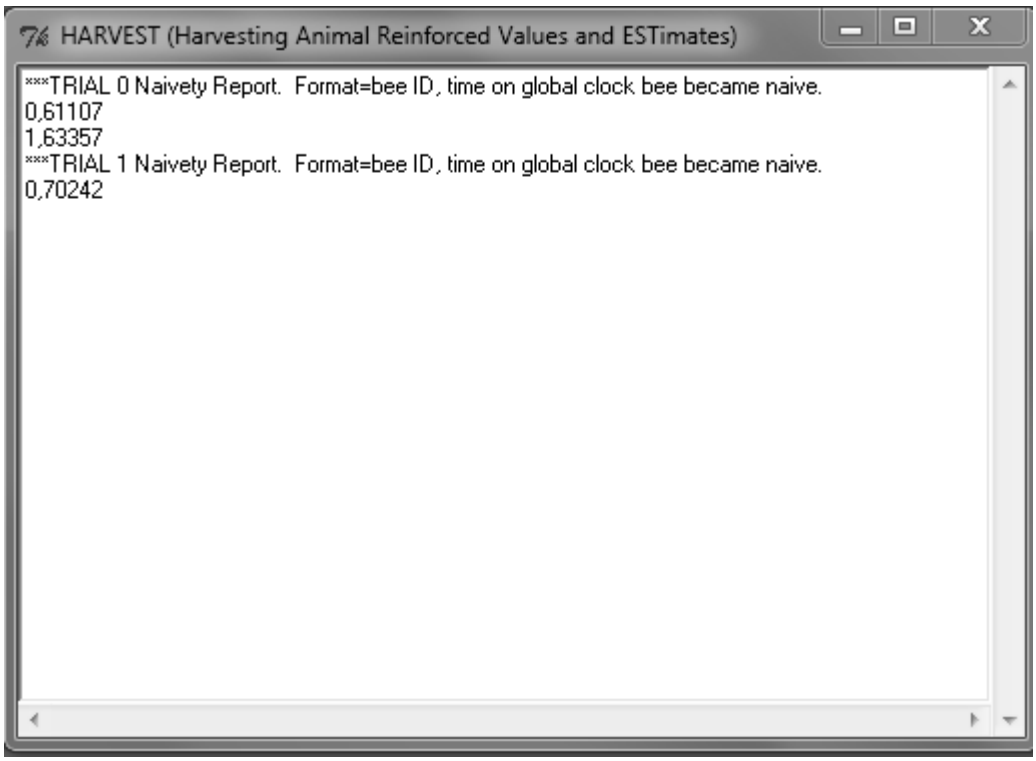

Figure C8. Window showing report of how often bees that returned to the nest became 'naive' via the Ω parameter (see Chapter Seven). Report shows sequential incidents, for each trial, with each incident presented as two comma separated values – the first indicating the ID number of the bee that became naive (first bee in simulation given bee ID of 0, second given ID of 1, and so on), and the second indicating the time on the global clock when the incident occurred. Accessed via 'View Naivety Report' button in main results window.

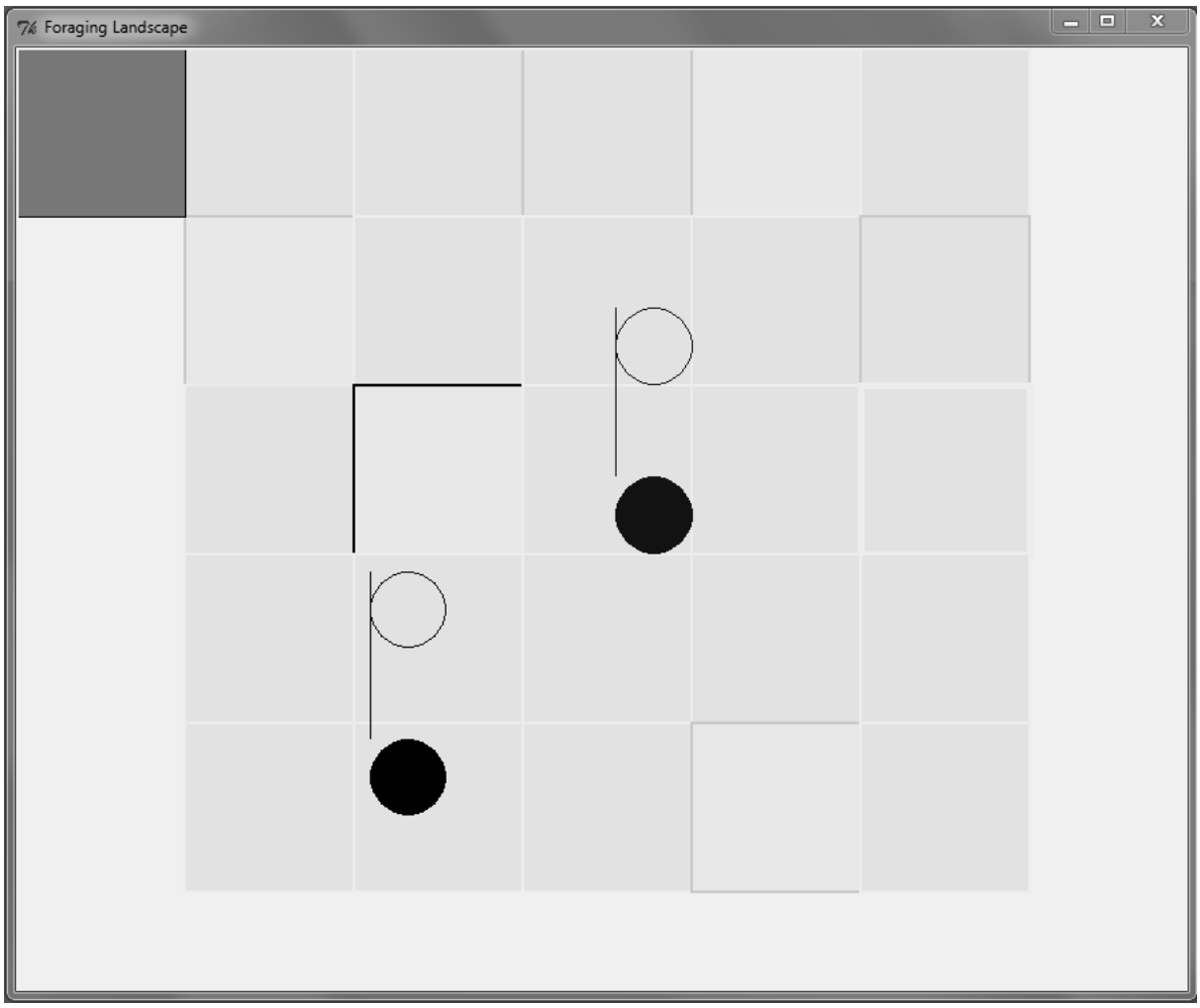

Figure C9. Window that allows the user to watch the movements that bees made within the landscape in the simulation. Patches are represented as green squares, whilst the nest is separate from the main matrix and is coloured brown. The shade of green in patches indicates their actual quality, with lighter shades indicating the patch has a lower quality, and darker shades indicating the patch has a higher quality. In depleting and replenishing landscapes, these shades can change over time. Patch with thicker border is the highest quality patch in the landscape (if there are other equally highest quality patches, they too will have a thicker border). The colour of the patch border indicates the accuracy of bee 0's estimate of the quality of the patch – accurate estimates are represented by cyan borders, underestimates are represented by yellow borders and overestimates are represented by black borders. Bees are represented as circles – closed circles indicate the current position of the bee, whilst open circles indicate where the bee has been previously. Lines indicate transitions between patches. The length of the 'tail' (the lines and open circles that collectively represent the history of a bee's movements) can be adjusted by the user in the main configuration window, as can the speed of the playback.

Appendix D : HARVEST Code Listing

HARVEST.py

```
# HARVEST

import random
#from pylab import *
import time
import math
from Tkinter import *

class GridPlayer:
   def playTour(self):
      root = Tk()
      root.title('Foraging Landscape')

      notAtEndOfTourFile = 1                  # 1 = NOT at end of tour file, 0 = at end of tour file
      tourRunning = 0                         # 1 = tour running, 0 = tour at an end (or not started)
      tempTourRead = ""                       # temporarily stores a line read in from tour file
      previousTempTourRead = []               # list that temporarily stores the previous tempTourRead, converted to int, for each bee, with element index corresponding to bee id
      previousOvalID = []                     # list that stores the ID number of the previous oval generated for each bee (marking the bee's previous position), with element index corresponding to bee id

      bestField = 0                           # stores the number of the best field in the landscape
      worstField = 0                          # stores the number of the worst field in the landscape

      totalPatches = 0                        # stores the total patches in the landscape
      gridTempFull = 0                        # temporarily stores the number of FULL flowers in a patch
      gridTempTotal = 0                       # temporarily stores the TOTAL number of flowers in a patch
      gridTempQuality = 0.0                   # temporarily stores FULL / TOTAL in a patch

      canvasWidth = 1                         # width of canvas in pixels
      canvasHeight = 1                        # height of canvas in pixels

      gridWidth = 1                           # number of cells wide (grid)
      gridHeight = 1                          # number of cells high (grid)

      resetX = 0                              # stores the x co-ordinate at which the grid of patches (non-nest) starts on the canvas

      topLeft = [0,0]                         # stores the top-left coordinates for drawing rectangles (grid cells)
      bottomRight = [0,0]                     # stores the bottom-right coordinates for drawing rectangles (grid cells)
      calculatedRectangleWidth = 0.0          # stores width of grid cells based on number of cells wide and canvas width
      calculatedRectangleHeight = 0.0         # stores height of grid cells based on number of cells high and canvas width

      nestID = 0                              # stores the grid ID (NOT THE PATCH ID as used by the rest of the code) of the nest

      tempCoords = []                         # temporarily stores coordinates, with element index corresponding to bee id
      destCoords = []                         # stores the coordinates of the destination location for each bee, with element index corresponding to bee id

      deletable = []                          # array that stores the IDs of the items that should be deleted if the tour is chosen to be run again

      stringOfQualities = ""                  # stores a string read in from patchQualitiesOverTime.txt
      listOfQualitiesAsStrings = []           # stores a list of strings generated from splitting stringOfQualities by a comma separation

      stringOfAccuracies = ""                 # stores a string read in from beeZeroEstimateAccuracy.txt
      listOfAccuraciesAsStrings = []          # stores a list of strings generated from splitting stringOfAccuracies by a comma separation

      beeInTransit = []                       # 1 = bee is in transit, 0 = bee is not in transit; element index corresponds to bee id

      visualDelay = 1.0                       # real-time length of time step on visual grid

      destNest = []                           # indicates that the destination is the nest; element index corresponds to bee id

      best = []                               # stores a list of all of the equally best fields in the landscape (in terms of quality)
      runningBest = 0.0                       # stores the highest quality in the landscape

      previousJourneyLine = []                # stores the ID of the previous journey line for each bee, with element index corresponding to bee id

      sizeOfColony = 0                        # stores the number of bees in the colony
      listOfTourFiles = []                    # stores a list of tour files

      colourTable = []                        # stores colour (as string) for each bee, with element index corresponding to bee id
```

```
    maxWidth = 0.0                  # stores the maximum width of a bee representation (when a single bee is simulated)
    maxHeight = 0.0                 # stores the maximum height of a bee representation (when a single bee is simulated)

    listOfFilesToBeConverted = []       # stores a list of tour files to generated converted copies
    previousReadConversion = ""         # stores the previously read in string for converting tour file
    lastReadConversion = ""             # stores the last-read in string for converting tour file
    notAtEndOfConversion = 1            # 1 = NOT at end of tour file during conversion process, 0 = at end of tour file
during conversion process

    gridTimer = 1                   # identifies the number of the cycle of visual updates on the grid; starts from 1
    allJourneyLines = []            # list stores lists in form [line id, time stamp], for all journey lines (for purposes of
fading)
    tempTimeStamp = 0               # temporarily stores a time stamp read in from allJourneyLines
    tempLineID = 0                  # temporarily stores a line id read in from allJourneyLines
    allTransparentOvals = []        # list stores lists in form [oval id, time stamp], for all ovals that became transparent (for
purposes of fading)
    tempOvalID = 0                  # temporarily stores an oval id read in from allTransparentOvals
    fadingInterval = 0              # stores the fading interval read in from file; specifies the number of visual update cycles
before a journey line / transparent oval should fade completely

    # read in the number of bees in the colony
    globalParametersFile = open('globalParametersFile.txt','r')
    globalParametersFile.readline()
    globalParametersFile.readline()
    globalParametersFile.readline()
    sizeOfColony = int(globalParametersFile.readline())
    globalParametersFile.close()

    # initialise lists
    for iy in range(0, sizeOfColony):
        previousTempTourRead.append(0)
        previousOvalID.append(0)
        destCoords.append([])
        previousJourneyLine.append(0)
        beeInTransit.append(0)
        destNest.append(0)
        tempCoords.append([])
        colourTable.append('')

    # populate the colour table
    for ik in range(0, sizeOfColony):
        if (ik % 10) == 0:
            colourTable[ik] = '#000000'
        if (ik % 10) == 1:
            colourTable[ik] = '#0000FF'
        if (ik % 10) == 2:
            colourTable[ik] = '#808080'
        if (ik % 10) == 3:
            colourTable[ik] = '#00FFFF'
        if (ik % 10) == 4:
            colourTable[ik] = '#800000'
        if (ik % 10) == 5:
            colourTable[ik] = '#FFFF00'
        if (ik % 10) == 6:
            colourTable[ik] = '#C0C0C0'
        if (ik % 10) == 7:
            colourTable[ik] = '#808000'
        if (ik % 10) == 8:
            colourTable[ik] = '#008080'
        if (ik % 10) == 9:
            colourTable[ik] = 'chocolate'

    # read in grid parameters from file and update parameters accordingly
    gridParametersFile = open('gridParametersFile.txt','r')

    gridParametersFile.readline()
    canvasWidth = int(gridParametersFile.readline())

    gridParametersFile.readline()
    canvasHeight = int(gridParametersFile.readline())

    gridParametersFile.readline()
    gridWidth = int(gridParametersFile.readline())

    gridParametersFile.readline()
    gridHeight = int(gridParametersFile.readline())
```

```
    gridParametersFile.readline()
    visualDelay = float(gridParametersFile.readline())

    gridParametersFile.readline()
    fadingInterval = int(gridParametersFile.readline())

    gridParametersFile.close()

    # GENERATE converted tour files for sequence analysis (copies are made, grid system still uses original tour files)
    for iz in range(0, sizeOfColony):
        listOfFilesToBeConverted.append((open('tourFiles/bee' + str(iz) + 'Tour.txt', 'r')))

        # create new file for conversion
        convertedCopy = open('convertedTours/convertedBee' + str(iz) + 'Tour.txt', 'w')
        # write header
        convertedCopy.write(str(listOfFilesToBeConverted[iz].readline()))
        convertedCopy.write(str(listOfFilesToBeConverted[iz].readline()))

        # whilst not at the end of the read file
        while notAtEndOfConversion == 1:
            lastReadConversion = listOfFilesToBeConverted[iz].readline()                # read the line

            # if read-in line differs from previous read-in line, and line is not a marker flag, write the line to the new file
            # and set the previous read = last read
            if lastReadConversion != previousReadConversion and "-" not in lastReadConversion:
                convertedCopy.write(str(lastReadConversion))
                previousReadConversion = lastReadConversion

            # flag if at end of file
            if lastReadConversion == "":
                notAtEndOfConversion = 0

        # close the read and write files
        convertedCopy.close()
        listOfFilesToBeConverted[iz].close()

        # reset the flag
        notAtEndOfConversion = 1

    canvas = Canvas(root, width=canvasWidth, height=canvasHeight)                # create a canvas widget

    canvas.pack(fill=BOTH, expand=YES)                # pack a widget into the parent Canvas widget; fill X and Y directions of
widget if widget grows, expand widget if parent size grows

    # calculate cell size
    calculatedRectangleWidth = int(((canvasWidth * 0.9) / gridWidth))
    calculatedRectangleHeight = int(((canvasHeight * 0.9) / gridHeight))

    # set value of resetX based on the width of a patch, and set initial top-left and bottom-right with incremented x-coordinates
    resetX = calculatedRectangleWidth
    topLeft = [resetX, 0]
    bottomRight = [resetX, 0]

    for ia in range(gridHeight):                # for each row
        bottomRight[1] = bottomRight[1] + calculatedRectangleHeight                # shift y of bottom right by 1 row
        for ib in range(gridWidth):                # for each column
            bottomRight[0] = bottomRight[0] + calculatedRectangleWidth                # shift x of bottom right by 1 column

            canvas.create_rectangle((topLeft[0], topLeft[1]),(bottomRight[0], bottomRight[1]), fill='#000000', width='2')        # draw
the rectangle

            topLeft[0] = bottomRight[0]            # set the x-coordinate of the top-left to the x-coordinate of the bottom-right (ie - shift
along by 1 column)

        topLeft[0] = resetX                                # once row done, reset top-left x to resetX
        topLeft[1] = topLeft[1] + calculatedRectangleHeight            # and shift y by 1 row

        bottomRight[0] = resetX                                # once row done, reset bottom-right x to resetX

    # READ IN NUMBER OF PATCHES IN LANDSCAPE
    dynamicLandscapeParametersFile = open('dynamicLandscapeParametersFile.txt','r')

    dynamicLandscapeParametersFile.readline()
    totalPatches = int(dynamicLandscapeParametersFile.readline())

    dynamicLandscapeParametersFile.close()

    # READ IN INITIAL LANDSCAPE QUALITIES AND CHANGE COLOUR OF CELLS TO REFLECT PATCH QUALITY
```

```
        initialResourceTable = open('initialResourceTable.txt', 'r')

        for ic in range(1, (totalPatches + 1)):
            initialResourceTable.readline()
            gridTempFull = int(initialResourceTable.readline())
            gridTempTotal = int(initialResourceTable.readline())

            gridTempQuality = float(gridTempFull) / float(gridTempTotal)

            if gridTempQuality == 0.0:
                canvas.itemconfig(ic, fill='#FFFFFF')
            if gridTempQuality > 0.0 and gridTempQuality <= 0.1:
                canvas.itemconfig(ic, fill='#EEFFEE')
            if gridTempQuality > 0.1 and gridTempQuality <= 0.2:
                canvas.itemconfig(ic, fill='#DDFFDD')
            if gridTempQuality > 0.2 and gridTempQuality <= 0.3:
                canvas.itemconfig(ic, fill='#CCFFCC')
            if gridTempQuality > 0.3 and gridTempQuality <= 0.4:
                canvas.itemconfig(ic, fill='#BBFFBB')
            if gridTempQuality > 0.4 and gridTempQuality <= 0.5:
                canvas.itemconfig(ic, fill='#AAFFBB')
            if gridTempQuality > 0.5 and gridTempQuality <= 0.6:
                canvas.itemconfig(ic, fill='#99FF99')
            if gridTempQuality > 0.6 and gridTempQuality <= 0.7:
                canvas.itemconfig(ic, fill='#88FF88')
            if gridTempQuality > 0.7 and gridTempQuality <= 0.8:
                canvas.itemconfig(ic, fill='#77FF77')
            if gridTempQuality > 0.8 and gridTempQuality <= 0.9:
                canvas.itemconfig(ic, fill='#66FF66')
            if gridTempQuality > 0.9 and gridTempQuality < 1.0:
                canvas.itemconfig(ic, fill='#55FF55')
            if gridTempQuality == 1.0:
                canvas.itemconfig(ic, fill='#44FF44')

        canvas.update()

        initialResourceTable.close()

        # draw the nest and store the grid ID
        nestID = tempTourRead = canvas.create_rectangle((0, 0),(calculatedRectangleWidth, calculatedRectangleHeight),
fill='#808000')

        # work out maximum width and height of bee representation
        maxWidth = 0.9 * calculatedRectangleWidth
        maxHeight = 0.9 * calculatedRectangleHeight

        tourRunning = 1                 # indicate tour run started

        # open the patchQualitiesOverTime file
        patchQualitiesOverTime = open('GraphData/patchQualitiesOverTime.txt','r')

        # open the beeZeroEstimateAccuracy file
        beeZeroEstimateAccuracy = open('GraphData/beeZeroEstimateAccuracy.txt','r')

        # READ IN TOUR FOR EACH BEE AND PLAY THE TOUR ON THE GRID
        # open all tour files
        for ir in range(0, sizeOfColony):
            listOfTourFiles.append((open('tourFiles/bee' + str(ir) + 'Tour.txt', 'r')))
            # skip header
            listOfTourFiles[ir].readline()
            listOfTourFiles[ir].readline()

        while tourRunning == 1:
            # read in the list of INITIAL patch qualities
            stringOfQualities = patchQualitiesOverTime.readline()

            # convert the string into a list of strings
            listOfQualitiesAsStrings = stringOfQualities.rsplit(',')

            # pop the last element from the list of strings (which is the string '\n')
            listOfQualitiesAsStrings.pop()

            # close then immediately reopen patch qualities file to put back at start
            patchQualitiesOverTime.close()
            patchQualitiesOverTime = open('GraphData/patchQualitiesOverTime.txt','r')

            # thin all borders that were previously thickened
            for ia in range(0, len(best)):
```

```
            if best[ia] < runningBest:
               canvas.itemconfig((ia + 1), width='2')

         best = []                              # reset (empty) the best list

         # make an element-for-element copy of listOfQualitiesAsStrings, but cast as floats
         for ib in range(0, len(listOfQualitiesAsStrings)):
            best.append(float(listOfQualitiesAsStrings[ib]))

         runningBest = float(max(best))         # find the highest quality in the landscape at this time

         # thicken all borders of cells corresponding to patches of equally best quality
         for ic in range(0, len(best)):
            if best[ic] == runningBest:
               canvas.itemconfig((ic + 1), width='5')

         canvas.update()

         # start the tour with the bees at the nest
         for ip in range(0, sizeOfColony):
            tempCoords[ip] = canvas.coords(nestID)
            previousOvalID[ip] = canvas.create_oval(((int(tempCoords[ip][0])+(0.1 * calculatedRectangleWidth)+float(ip * (maxWidth / sizeOfColony))),(int(tempCoords[ip][1])+(0.1 * calculatedRectangleHeight)+float(ip * (maxHeight / sizeOfColony)))),(((int(tempCoords[ip][0])+(0.1 * calculatedRectangleWidth)+float(ip * (maxWidth / sizeOfColony))) + float(maxWidth/sizeOfColony)),((int(tempCoords[ip][1])+(0.1 * calculatedRectangleHeight)+float(ip * (maxHeight / sizeOfColony)))+(maxHeight/sizeOfColony))),fill=str(colourTable[ip]),outline=str(colourTable[ip]))
            deletable.append(previousOvalID[ip])
            canvas.update()

         while notAtEndOfTourFile == 1:
            for iFile in range(0, len(listOfTourFiles)):
               tempTourRead = listOfTourFiles[iFile].readline()

               # flag if end of file reached
               if tempTourRead == "":
                  notAtEndOfTourFile = 0
               else:
                  tempTourRead = int(tempTourRead)          # convert to int

                  if tempTourRead != previousTempTourRead[iFile]:          # if bee has moved
                     canvas.itemconfig(previousOvalID[iFile], fill='')     # make the current oval transparent
                     allTransparentOvals.append([previousOvalID[iFile], gridTimer])  # append oval id and time stamp to all transparent ovals list

                     # if tour file indicates a (destination = non-nest) move has been instigated
                     if tempTourRead < 0 and tempTourRead != -9999999 and tempTourRead != -8888888:
                        tempTourRead = int(math.sqrt(math.pow(tempTourRead,2)))     # get rid of the - sign

                        destCoords[iFile] = canvas.coords(tempTourRead)          # find the destination coordinates

                        # draw a line between the source and the destination
                        previousJourneyLine[iFile] = (canvas.create_line(((int(tempCoords[iFile][0])+(0.1 * calculatedRectangleWidth)+float(iFile * (maxWidth / sizeOfColony))),(int(tempCoords[iFile][1])+(0.1 * calculatedRectangleHeight)+float(iFile * (maxHeight / sizeOfColony)))),((int(destCoords[iFile][0])+(0.1 * calculatedRectangleWidth)+float(iFile * (maxWidth / sizeOfColony))),(int(destCoords[iFile][1])+(0.1 * calculatedRectangleHeight)+float(iFile * (maxHeight / sizeOfColony)))),fill=str(colourTable[iFile])))
                        allJourneyLines.append([previousJourneyLine[iFile], gridTimer])      # append id and time stamp to all journey lines list
                        deletable.append(previousJourneyLine[iFile])
                        canvas.itemconfig(previousJourneyLine[iFile], width='2')
                        canvas.update()
                        beeInTransit[iFile] = 1                  # flag that bee is in transit

                        # if the destination is NOT the nest
                        if destNest[iFile] == 0:
                           # skip a line to allow for the fact that journey is reported in tour with (distance in time units) old patch entries preceeded by a -dest entry
                           # this skip accomodates the -dest entry
                           listOfTourFiles[iFile].readline()
                        else:
                           # if destination IS the nest, don't skip the line but indicate that the destination is no longer the nest
                           destNest[iFile] = 0

                     # if tour file indicates a field-to-nest move has been instigated
                     if tempTourRead == -8888888:
                        destNest[iFile] = 1                  # set flag to indicate destination = nest

                        destCoords[iFile] = canvas.coords(nestID)          # find the destination coordinates
```

```
                    # draw a line between the source and the destination
                    previousJourneyLine[iFile] = (canvas.create_line(((int(tempCoords[iFile][0])+(0.1 *
calculatedRectangleWidth)+float(iFile * (maxWidth / sizeOfColony))),(int(tempCoords[iFile][1])+(0.1 *
calculatedRectangleHeight)+float(iFile * (maxHeight / sizeOfColony)))),((int(destCoords[iFile][0])+(0.1 *
calculatedRectangleWidth)+float(iFile * (maxWidth / sizeOfColony))),(int(destCoords[iFile][1])+(0.1 *
calculatedRectangleHeight)+float(iFile * (maxHeight / sizeOfColony)))),fill=str(colourTable[iFile])))
                    allJourneyLines.append([previousJourneyLine[iFile], gridTimer])  # append id and time stamp to all
journey lines list
                    deletable.append(previousJourneyLine[iFile])
                    canvas.itemconfig(previousJourneyLine[iFile], width='2')
                    canvas.update()
                    beeInTransit[iFile] = 1                    # flag that bee is transit

                    # skip a line to allow for the fact that journey is reported in tour with (distance in time units) old patch entries
preceeded by a -dest entry
                    # this skip accomodates the -dest entry
                    listOfTourFiles[iFile].readline()

                # if tour file indicates the journey has come to an end
                if tempTourRead == -9999999:
                    tempCoords[iFile] = destCoords[iFile]                    # current coordinates = destination
coordinates
                    previousOvalID[iFile] = canvas.create_oval(((int(tempCoords[iFile][0])+(0.1 *
calculatedRectangleWidth)+float(iFile * (maxWidth / sizeOfColony))),(int(tempCoords[iFile][1])+(0.1 *
calculatedRectangleHeight)+float(iFile * (maxHeight / sizeOfColony)))),(((int(tempCoords[iFile][0])+(0.1 *
calculatedRectangleWidth)+float(iFile * (maxWidth / sizeOfColony))) +
float(maxWidth/sizeOfColony)),((int(tempCoords[iFile][1])+(0.1 * calculatedRectangleHeight)+float(iFile * (maxHeight /
sizeOfColony)))+(maxHeight/sizeOfColony))),fill=str(colourTable[iFile]))  # draw a filled oval at the new location (now the
current location)
                    deletable.append(previousOvalID[iFile])
                    canvas.itemconfig(previousJourneyLine[iFile], width='1')
                    canvas.update()
                    beeInTransit[iFile] = 0                    # flag that bee is no longer in transit

                    # if the destination is NOT the nest
                    if destNest[iFile] == 0:
                        # skip a line to allow for the fact that there is an additional old dest entry in the tour file after the -9999999
marker
                        listOfTourFiles[iFile].readline()

                previousTempTourRead[iFile] = tempTourRead  # update the last read-in from the tour file

            canvas.update()

        # read in the list of patch qualities for this time unit
        stringOfQualities = patchQualitiesOverTime.readline()

        # convert the string into a list of strings
        listOfQualitiesAsStrings = stringOfQualities.rsplit(',')

        # pop the last element from the list of strings (which is the string '\n')
        listOfQualitiesAsStrings.pop()

        # for each patch, update the colours in the grid appropriately (item id is iq+1 as item ids start from 1, whilst patch 1 is
        # element iq = 0
        for iq in range(0, len(listOfQualitiesAsStrings)):
            if float(listOfQualitiesAsStrings[iq]) == 0.0:
                canvas.itemconfig((iq + 1), fill='#FFFFFF')
            if float(listOfQualitiesAsStrings[iq]) > 0.0 and gridTempQuality <= 0.1:
                canvas.itemconfig((iq + 1), fill='#EEFFEE')
            if float(listOfQualitiesAsStrings[iq]) > 0.1 and gridTempQuality <= 0.2:
                canvas.itemconfig((iq + 1), fill='#DDFFDD')
            if float(listOfQualitiesAsStrings[iq]) > 0.2 and gridTempQuality <= 0.3:
                canvas.itemconfig((iq + 1), fill='#CCFFCC')
            if float(listOfQualitiesAsStrings[iq]) > 0.3 and gridTempQuality <= 0.4:
                canvas.itemconfig((iq + 1), fill='#BBFFBB')
            if float(listOfQualitiesAsStrings[iq]) > 0.4 and gridTempQuality <= 0.5:
                canvas.itemconfig((iq + 1), fill='#AAFFBB')
            if float(listOfQualitiesAsStrings[iq]) > 0.5 and gridTempQuality <= 0.6:
                canvas.itemconfig((iq + 1), fill='#99FF99')
            if float(listOfQualitiesAsStrings[iq]) > 0.6 and gridTempQuality <= 0.7:
                canvas.itemconfig((iq + 1), fill='#88FF88')
            if float(listOfQualitiesAsStrings[iq]) > 0.7 and gridTempQuality <= 0.8:
                canvas.itemconfig((iq + 1), fill='#77FF77')
            if float(listOfQualitiesAsStrings[iq]) > 0.8 and gridTempQuality <= 0.9:
                canvas.itemconfig((iq + 1), fill='#66FF66')
            if float(listOfQualitiesAsStrings[iq]) > 0.9 and gridTempQuality < 1.0:
```

```
                canvas.itemconfig((iq + 1), fill='#55FF55')
            if float(listOfQualitiesAsStrings[iq]) == 1.0:
                canvas.itemconfig((iq + 1), fill='#44FF44')

        # thin all borders that were previously thickened
        for ia in range(0, len(best)):
            if best[ia] < runningBest:
                canvas.itemconfig((ia + 1), width='2')

        best = []                               # reset (empty) the best list

        # make an element-for-element copy of listOfQualitiesAsStrings, but cast as floats
        for ib in range(0, len(listOfQualitiesAsStrings)):
            best.append(float(listOfQualitiesAsStrings[ib]))

        if len(best) > 0:                       # if best is not empty
            runningBest = float(max(best))          # find the highest quality in the landscape at this time

            # thicken all borders of cells corresponding to patches of equally best quality
            for ic in range(0, len(best)):
                if best[ic] == runningBest:
                    canvas.itemconfig((ic + 1), width='5')

        # read in the list of bee zero accuracies for this time unit
        stringOfAccuracies = beeZeroEstimateAccuracy.readline()

        # convert the string into a list of strings
        listOfAccuraciesAsStrings = stringOfAccuracies.rsplit(',')

        # pop the last element from the list of strings (which is the string '\n')
        listOfAccuraciesAsStrings.pop()

        # for each patch, update the border colour according to whether bee zero's estimate is above the actual, below the actual or
accurate
        for ir in range(0, len(listOfAccuraciesAsStrings)):
            if int(listOfAccuraciesAsStrings[ir]) == 0:
                canvas.itemconfig((ir + 1), outline='cyan')
            if int(listOfAccuraciesAsStrings[ir]) == -1:
                canvas.itemconfig((ir + 1), outline='#FFFF00')
            if int(listOfAccuraciesAsStrings[ir]) == 1:
                canvas.itemconfig((ir + 1), outline='#000000')

        # if bee is not in transit, re-fill the oval representing bee (this is done due to anomaly
        # where the oval becomes transparent when the bee is foraging with grid patch quality updates
        for iu in range(0, sizeOfColony):
            if beeInTransit[iu] == 0:
                canvas.itemconfig(previousOvalID[iu], fill=str(colourTable[iu]), outline=str(colourTable[iu]))
                canvas.update()

        # check through all journey lines list, and update any lines that need fading
        for ia in range(0, len(allJourneyLines)):
            tempLineID = allJourneyLines[ia][0]
            tempTimeStamp = allJourneyLines[ia][1]

            if (gridTimer - tempTimeStamp) >= fadingInterval:
                #canvas.itemconfig(tempLineID, fill="#FFFFFF")      # use this to fade to white
                canvas.delete(tempLineID)                       # use this to delete entirely

            canvas.update()

        # check through all transparent ovals list, and update any ovals that need fading
        for iz in range(0, len(allTransparentOvals)):
            tempOvalID = allTransparentOvals[iz][0]
            tempTimeStamp = allTransparentOvals[iz][1]

            if (gridTimer - tempTimeStamp) >= fadingInterval:
                #canvas.itemconfig(tempOvalID, outline="#FFFFFF")   # use this to fade to white
                canvas.delete(tempOvalID)                       # use this to delete entirely

            canvas.update()

        time.sleep(visualDelay)                 # sleep for specified delay

        gridTimer = gridTimer + 1               # increment the grid timer

    # close the files
    for ib in range(0, len(listOfTourFiles)):
        listOfTourFiles[ib].close()
```

```
        # open the files again
        for ie in range(0, len(listOfTourFiles)):
            listOfTourFiles[ie] = open('tourFiles/bee' + str(ie) + 'Tour.txt', 'r')
            # skip header
            listOfTourFiles[ie].readline()
            listOfTourFiles[ie].readline()

        print "Tour Completed."
        raw_input("PRESS ENTER TO VIEW TOUR AGAIN...")

        # reset the grid timer
        gridTimer = 1

        # empty the lists used for fading
        allJourneyLines = []
        allTransparentOvals = []

        # delete each item marked for deletion, ready for the tour to be re-played
        for iz in deletable:
            canvas.delete(iz)

        deletable = []                          # reset the list

        notAtEndOfTourFile = 1                          # reset the flag

        # READ IN AGAIN INITIAL LANDSCAPE QUALITIES AND CHANGE COLOUR OF CELLS TO REFLECT PATCH
QUALITY
        initialResourceTable = open('initialResourceTable.txt', 'r')

        for ic in range(1, (totalPatches + 1)):
            initialResourceTable.readline()
            gridTempFull = int(initialResourceTable.readline())
            gridTempTotal = int(initialResourceTable.readline())

            gridTempQuality = float(gridTempFull) / float(gridTempTotal)

            if gridTempQuality == 0.0:
                canvas.itemconfig(ic, fill='#FFFFFF')
            if gridTempQuality > 0.0 and gridTempQuality <= 0.1:
                canvas.itemconfig(ic, fill='#EEFFEE')
            if gridTempQuality > 0.1 and gridTempQuality <= 0.2:
                canvas.itemconfig(ic, fill='#DDFFDD')
            if gridTempQuality > 0.2 and gridTempQuality <= 0.3:
                canvas.itemconfig(ic, fill='#CCFFCC')
            if gridTempQuality > 0.3 and gridTempQuality <= 0.4:
                canvas.itemconfig(ic, fill='#BBFFBB')
            if gridTempQuality > 0.4 and gridTempQuality <= 0.5:
                canvas.itemconfig(ic, fill='#AAFFBB')
            if gridTempQuality > 0.5 and gridTempQuality <= 0.6:
                canvas.itemconfig(ic, fill='#99FF99')
            if gridTempQuality > 0.6 and gridTempQuality <= 0.7:
                canvas.itemconfig(ic, fill='#88FF88')
            if gridTempQuality > 0.7 and gridTempQuality <= 0.8:
                canvas.itemconfig(ic, fill='#77FF77')
            if gridTempQuality > 0.8 and gridTempQuality <= 0.9:
                canvas.itemconfig(ic, fill='#66FF66')
            if gridTempQuality > 0.9 and gridTempQuality < 1.0:
                canvas.itemconfig(ic, fill='#55FF55')
            if gridTempQuality == 1.0:
                canvas.itemconfig(ic, fill='#44FF44')

        canvas.update()

        initialResourceTable.close()

        # close pacthQualitiesOverTime then immediately re-open to put back at start of file
        patchQualitiesOverTime.close()
        patchQualitiesOverTime = open('GraphData/patchQualitiesOverTime.txt','r')

        # close beeZeroEstimateAccuracy then immediately re-open to put back at start of file
        beeZeroEstimateAccuracy.close()
        beeZeroEstimateAccuracy = open('GraphData/beeZeroEstimateAccuracy.txt','r')

class Nest:
    nestNectar = 0
    nestTimeExisted = 0
    nestNectarAccumulationRate = 0
```

```
    previousNestNectarAccumulationRate = 0
    nestRateState = 1

    def initialSetup (self):
        self.nestNectar = 0
        self.nestTimeExisted = 0
        self.nestNectarAccumulationRate = 0
        self.previousNestNectarAccumulationRate = 0
        self.nestRateState = 1

    def getNestNectar (self):
        return self.nestNectar

    def getNestTimeExisted (self):
        return self.nestTimeExisted

    def getNestNectarAccumulationRate (self):
        return self.nestNectarAccumulationRate

    def getPreviousNestNectarAccumulationRate (self):
        return self.previousNestNectarAccumulationRate

    def getNestRateState (self):
        return self.nestRateState

    def accumulateNectar (self, additionalAmount):
        self.nestNectar = self.nestNectar + additionalAmount

    def advanceTime (self, temporalAdvance):
        self.nestTimeExisted = self.nestTimeExisted + temporalAdvance

    def updateRateState (self, newState):
        self.nestRateState = newState

    def calculateNestNectarAccumulationRate (self):
        self.previousNestNectarAccumulationRate = self.nestNectarAccumulationRate
        self.nestNectarAccumulationRate = self.nestNectar / self.nestTimeExisted

class Bee:
    completedOneBout = 0

    beeID = 0                           # stores the ID number of the bee
    beeCreationTime = 0                 # stores the time at which the bee was created

    greedyLifeCount = 0                 # tallies greedy decisions across life
    nnGreedyLifeCount = 0               # tallies not-necc greedy decisions across life
    dylFieldForages = []                # stores list with each element representing a tally of the number of forages (by this bee) in
the field indicated by the element's index
    traffic = []                        # lists qty of transitions in each bout
    performance = []                    # lists bout performance scores
    allTours = []                       # list of tours across life
    tour = []                           # stores the tour for the current bout
    allToursMerged = []                 # lists every lcoation of the bee over time as a constant single list of locations

    lifespan = 0                        # lifespan of bee in foraging bouts
    samplingInterval = 0                # time to sample a single flower
    foragingBoutTally = 0               # tallies number of foraging bouts conducted
    fixedNectarCapacity = 0             # total unchanging nectar capacity of the bee
    epsilon = 0                         # epsilon value - prob of choosing not-neccesarily greedily
    alpha = 0                           # identifies proportion of newly calculated action value to be used in replacement
    beta = 0                            # caution parameter - trust in bee's new sampling experience
    plusRate = 0                        # CURRENTLY UNUSED - proportion of initial action value to increase if applicable
    minusRate = 0                       # CURRENTLY UNUSED - proportion of initial action value to decrease if applicable
    requiredReduction = 0               # temporarily stores the amount that the bee's remaining nectar capacity should reduce
    totalDecisionsMade = 0              # tallies the total number of decisions (of all types) made across the bee's lifetime
    dylBoutsOnlyVisited = []            # stores list with each element representing a tally of the number of foraging bouts where
the bee only visits the field indicated by the element's index
    fullFlowerSelections = 0            # number of times a FULL flower is sampled across the bee's lifetime
    emptyFlowerSelections = 0           # number of times an EMPTY flower is sampled across the bee's lifetime

    distanceMatrix = []                 # in format [[dist(0,0), dist(0,1)...],[dist(1,0)...]...]
    estimatedQualityTable = []          # stores the bee's estimated quality of each field (with fields indicated by the indices)

    emptyTable = []                     # empty list used for valid set up
    dylActionValueTable = []            # stores list with each element representing list of action-values for all valid actions from
the field indicated by the main element's index

    boutPerformance = 0                 # stores performance score for the bout
```

```
    boutTimeLifeStore = []                    # stores list of 'bout time taken's

    decisionQtyRecord = []                    # list of decision quantities in each bout

    state = "AT NEST"                         # the state of the bee (details its current activity)
    completionTimeOfState = 0                 # the time (on the global clock) at which the current state of the bee will complete
    actionChoiceType = "greedy"               # the policy of the bee (way of choosing action value)
    decisionsInBout = 0                       # tallies the number of decisions made in a foraging bout
    currentField = 0                          # logs the current location of the bee
    destinationField = -1                     # logs the proposed new location of the bee (and the new moved-to location of the bee until it has foraged)
    chosenCell = [0, 0]                       # stores the coordinates of the chosen action value in the form [main element index, nested element index]
    trafficCount = 0                          # tallies number of transitions (NOT stays) in a foraging bout
    tempStorePreviousQualityEstimate = 0.0    # temporarily stores the previous estimated quality of a field
    tempStoreFullFlowers = 0.0                # temporarily stores a read-in number of FULL flowers
    tempStoreTotalFlowers = 0.0               # temporarily stores a read-in number of Total flowers
    flowerTypeToSample = "EMPTY"              # stores the flower type that the bee will encounter, as chosen by the flower selection function
    remainingNectarCapacity = 0               # the remaining amount of nectar that can be carried during a foraging bout
    full = 'false'                            # records if the bee is full to capacity with nectar
    nectarExtractedInBout = 0                 # records the amount of nectar the bee extracted (or has so far extracted) in a bout
    timeStartedBout = 0                       # records the time on the global clock when a foraging bout was started
    timeFinishedBout = 0                      # records the time on the global clock when a foraging bout was completed
    rateOfUptakeOverTime = []                 # list of rate of nectar uptake values for each tick of the global clock since the bee's existence
    totalNectarCollectedOverLife = 0.0        # records the total amount of nectar collected over the bee's life
    dylEstimatedQualityHistory = []           # stores list with each element representing a list of estimated qualities over time for the field indicated by the main element's index
    remainingCapacityHistory = []             # stores the remaining capacity of the bee over time
    boutCompleted = 'false'                   # flag to indicate the end of a bout
    dylMeanEstimatedQualityHistory = []       # stores list with each element representing a list of mean estimated qualities over time for the field indicated by the main element's index
    tempMeanEstCalc = 0.0                     # temporarily stores a calculation of mean estimated quality

    dylFieldVisitsPerBout = []                # stores list with each element representing a list of visitation numbers (per bout) over time in the field indicated by the main element's index

    dylFieldBoutCount = []                    # stores list with each element representing a tally of the number of visits (in a bout) to the field indicated by the element's index

    dylFieldVisitedInLife = []                # stores list with each element being 0 if field not visited yet by the bee in the trial, or 1 if field has been visited at least once in the trial.  Field indicated by the element's index
    dylFieldVisitedInBout = []                # stores list with each element being 0 if field not visited yet by the bee in the foraging bout, or 1 if field has been visited at least once in the foraging bout.  Field indicated by the element's index

    listOfUniquesPerBout = []                 # stores list with each element representing the number of unique fields visited in the foraging bout indicated by the element's index

    explicitFlowerToSample = [0,0,0]          # stores the field, the explicit flower and the reward of the flower selected by the bee

    initialActionValueCalculationMade = 0     # 0 = initial action value calculation not made for the bee, 1 = has been made for the bee

    residencePerPatch = []                    # list which stores sub-lists of the form [residence (in flower visits) in patch, patch, patch type] for every visit to a patch (starts on arrival, ends on departure)
    runningResidenceCounter = 0               # keeps a running tally of flower visits for this bee in the current patch

    # initialisation function
    def initialSetup(self, inputLifespan, inputSamplingInterval, inputfixedNectarCapacity, inputEpsilon, inputAlpha, inputBeta, inputPlusRate, inputMinusRate, inputActionValueTable, inputDistanceMatrix, inputEstimatedQualityTable, inputNestRateState):
        self.completedOneBout = 0

        self.beeID = 0
        self.beeCreationTime = 0

        self.greedyLifeCount = 0
        self.nnGreedyLifeCount = 0
        self.dylFieldForages = []
        self.traffic = []
        self.performance = []
        self.allTours = []
        self.tour = []
        self.allToursMerged = []

        self.lifespan = inputLifespan
        self.samplingInterval = inputSamplingInterval
        self.foragingBoutTally = 0
```

```
        self.fixedNectarCapacity = inputfixedNectarCapacity
        self.epsilon = inputEpsilon
        self.alpha = inputAlpha
        self.beta = inputBeta
        self.plusRate = inputPlusRate
        self.minusRate = inputMinusRate
        self.requiredReduction = 0
        self.totalDecisionsMade = 0
        self.dylBoutsOnlyVisited = []
        self.fullFlowerSelections = 0
        self.emptyFlowerSelections = 0

        self.distanceMatrix = inputDistanceMatrix
        self.estimatedQualityTable = inputEstimatedQualityTable                    # perceived quality table - updated from central file for field x when visiting field x

        self.emptyTable = []
        self.dylActionValueTable = inputActionValueTable

        self.boutPerformance = 0
        self.boutTimeLifeStore = []

        self.decisionQtyRecord = []

        self.state = "AT NEST"
        self.completionTimeOfState = 0
        self.actionChoiceType = "greedy"
        self.decisionsInBout = 0
        self.currentField = 0
        self.destinationField = -1
        self.chosenCell = [0, 0]
        self.trafficCount = 0
        self.tempStorePreviousQualityEstimate = 0.0
        self.tempStoreFullFlowers = 0.0
        self.tempStoreTotalFlowers = 0.0
        self.flowerTypeToSample = "EMPTY"
        self.remainingNectarCapacity = 0
        self.full = 'false'
        self.nectarExtractedInBout = 0
        self.timeStartedBout = 0
        self.timeFinishedBout = 0
        self.rateOfUptakeOverTime = []
        self.totalNectarCollectedOverLife = 0.0
        self.dylEstimatedQualityHistory = []
        self.remainingCapacityHistory = []                # stores the remaining capacity of the bee over time
        self.boutCompleted = 'false'                      # flag to indicate the end of a bout
        self.dylMeanEstimatedQualityHistory = []
        self.tempMeanEstCalc = 0.0

        self.dylFieldVisitsPerBout = []
        self.dylFieldBoutCount = []

        self.dylFieldVisitedInLife = []
        self.dylFieldVisitedInBout = []

        self.listOfUniquesPerBout = []

        self.explicitFlowerToSample = [0,0,0]

        self.initialActionValueCalculationMade = 0

        self.residencePerPatch = []
        self.runningResidenceCounter = 0

        # INITIALISATION of certain lists
        for dylIX in range(0, len(self.estimatedQualityTable)):
            self.dylFieldForages.append(0)
            self.dylBoutsOnlyVisited.append(0)
            self.dylFieldBoutCount.append(0)
            self.dylFieldVisitedInLife.append(0)
            self.dylFieldVisitedInBout.append(0)
            self.dylEstimatedQualityHistory.append([])
            self.dylMeanEstimatedQualityHistory.append([])
            self.dylFieldVisitsPerBout.append([])

    def choosePolicyType(self):
        greedAssessment = random.uniform(0.0,1.0)          # generate a random number to decide if bee should be greedy or not-necessarily greedy
```

```
        if greedAssessment < (1 - self.epsilon):
            self.actionChoiceType = "greedy"                # records choice of action will be greedy
            ##print "@@@  BEE ID : " + str(self.beeID)
                ##print "Random Number Generated = " + str(round(greedAssessment, 3)) + "   (< " + str(round(1 - self.epsilon, 3)) +
")    **A GREEDY CHOICE WILL BE MADE"
                ##print "--------------------------------"
            self.greedyLifeCount = self.greedyLifeCount + 1                     # increment greedy decisions tally
            self.totalDecisionsMade = self.totalDecisionsMade + 1               # increment total decisions tally
        else:
            if greedAssessment >= (1 - self.epsilon):
                self.actionChoiceType = "nnGreedy"          # records choice of action will be not-necessarily greedy
                ##print "@@@  BEE ID : " + str(self.beeID)
                ##print "Random Number Generated = " + str(round(greedAssessment, 3)) + "   (>= " + str(round(1 - self.epsilon, 3)) + ")
**A not-necessarily greedy CHOICE WILL BE MADE"
                ##print "--------------------------------"
                self.nnGreedyLifeCount = self.nnGreedyLifeCount + 1             # increment not-necessarily greedy decisions tally
                self.totalDecisionsMade = self.totalDecisionsMade + 1           # increment total decisions tally
            else:
                ##print "***** INVALID ACTION TYPE SELECTION *****"                # flags any invalid decision types
                self.totalDecisionsMade = self.totalDecisionsMade + 1           # increment total decisions tally

        self.decisionsInBout = self.decisionsInBout + 1         # increment decisions IN THIS BOUT tally

    def selectActionValue(self):
        stepToBtr = []                          # temporary 'stepping-stone' list
        stepToAllRandomList = []                # temporary 'stepping-stone' list
        btrListMin = []                         # stores identically low action values (and their column coordinates) to break ties randomly
        chosenActionValue = 0                   # stores the chosen action value
        chosenActionValuePacked = []            # stores the chosen action value along with its coordinates in the relevant row of the
action value table
        tempTooBigIndex = 0                     # stores the randomly generated column coordinate of a selected action value when
action values are too high (beyond the range of float)
        allOpenRandom = []                      # stores ALL selectable action values (and their column coordinates) to choose when
not-necessarily greedy

        # if choice is to be greedy, then grab a list of all equally low action values in the relevant row, and randomly choose one
        # - this enforces the 'BTR' condition
        if self.actionChoiceType == "greedy":
            stepToBtr = self.dylActionValueTable[self.currentField]         # grab the correct row (based on the current field)

            # construct list of equally lowest values in the grabbed row, with their indices (but not with dest = 0 (the NEST))
            for dylIA,dylIB in enumerate(stepToBtr):
                if dylIB == min(stepToBtr[1:len(stepToBtr)]):
                    if dylIA != 0:
                        btrListMin.append([dylIB,dylIA])

            #if self.beeID == 0:
                ##print "Source = " + str(self.currentField)
                ##print str(stepToBtr)

            # randomly choose one of the 'minimums' from the grabbed row, and store a)the action-value itself and b)the coordinates of
the action value
            if len(btrListMin) > 0:
                chosenActionValuePacked = random.choice(btrListMin)
                chosenActionValue = chosenActionValuePacked[0]
                self.chosenCell = [self.currentField, chosenActionValuePacked[1]]

                #if self.beeID == 0:
                    ##print "Chose Dest = " + str(chosenActionValuePacked[1])
            else:                                       # catches instances where action value is beyond range of float
                tempTooBigIndex = random.choice(range(0,len(stepToBtr)))        # choose a random valid column
                self.chosenCell = [self.currentField, tempTooBigIndex]

        # if choice is to be not-necessarily greedy, then grab a list of ALL action values in the relevant row
        # and randomly choose one
        if self.actionChoiceType == "nnGreedy":
            stepToAllRandomList = self.dylActionValueTable[self.currentField]    # grab the correct row (based on the current field)

            # construct list of all action values in the grabbed row, with their indices (but not with dest = 0 (the NEST))
            for dylIC, dylID in enumerate(stepToAllRandomList):
                if dylIC != 0:
                    allOpenRandom.append([dylID, dylIC])

            # randomly choose one of the action-values, and store a)the action-value itself and b)the coordinates of the action-value
            chosenActionValuePacked = random.choice(allOpenRandom)
            chosenActionValue = chosenActionValuePacked[0]
            self.chosenCell = [self.currentField, chosenActionValuePacked[1]]
```

```
        # the destination field is identified from the chosen action value, and the relevant move tallies
        # are incremented
        self.destinationField = self.chosenCell[1]
        self.dylFieldForages[self.destinationField] = self.dylFieldForages[self.destinationField] + 1
        self.dylFieldBoutCount[self.destinationField] = self.dylFieldBoutCount[self.destinationField] + 1

        # if the bee is to move from its current location, then the traffic tally is incremented (unless its a NEST - field move)
        if self.destinationField != self.currentField:
            if self.currentField != 0:
                self.trafficCount = self.trafficCount + 1

        ##print "@@@ BEE ID : " + str(self.beeID)
        ##print str(self.currentField) + " --> A    " + str(round((self.chosenCell[0])[0], 3))
        ##print str(self.currentField) + " --> B    " + str(round((self.chosenCell[0])[1], 3))
        ##print str(self.currentField) + " --> C    " + str(round((self.chosenCell[0])[2], 3))
        ##print "Policy = " + str(self.actionChoiceType)
        ##print "Selected Action Value = " + str(round(chosenActionValue, 3))
        ##print ""

    def selectFlower(self, explicitFlowerAllFields):
        tempExplicitFlowerChoice = 0                    # temporarily stores the chosen flower index
        numberOfFullsHere = 0                           # temporarily stores the number of FULL flowers in the patch

        # count the number of full flowers in the patch
        numberOfFullsHere = int(explicitFlowerAllFields[self.destinationField].count(1))

        # if there are no full flowers at this time, change the estimated quality of the patch to near 0 (not exactly 0 to avoid divide by 0
errors)
        if numberOfFullsHere == 0:
            self.estimatedQualityTable[self.destinationField] = 0.00000000000000000001

        # we store Estimated Quality (x) OLD (the 'before sampling' estimate of the
        # quality level of the field from which the bee is foraging)
        self.tempStorePreviousQualityEstimate = self.estimatedQualityTable[self.destinationField]
        ##print "@@@ BEE ID : " + str(self.beeID)
        ##print "     --> Previous Estimated Quality (" + str(self.destinationField) + ") = " +
str(round(self.tempStorePreviousQualityEstimate, 3))

        # randomly choose a flower from the array for the chosen field
        tempExplicitFlowerChoice = int(random.randint(0, (len(explicitFlowerAllFields[self.destinationField]) - 1)))
        self.explicitFlowerToSample[0] = int(self.destinationField)
        self.explicitFlowerToSample[1] = int(tempExplicitFlowerChoice)
        self.explicitFlowerToSample[2] = int(explicitFlowerAllFields[self.destinationField][tempExplicitFlowerChoice])

        # identify whether chosen flower is FULL or empty, and update appropriate variables and tallies
        if self.explicitFlowerToSample[2] == 1:
            self.flowerTypeToSample = "FULL"
            self.fullFlowerSelections = self.fullFlowerSelections + 1
        else:
            self.flowerTypeToSample = "EMPTY"
            self.emptyFlowerSelections = self.emptyFlowerSelections + 1

    def extractNectar(self):
        # remaining nectar capacity of bee reduced according to the flower sampled
        if self.flowerTypeToSample == "FULL":
            self.requiredReduction = 1.0
        if self.flowerTypeToSample == "EMPTY":
            self.requiredReduction = 0.0

        # apply the relevant reduction
        self.remainingNectarCapacity = self.remainingNectarCapacity - self.requiredReduction

        self.nectarExtractedInBout = self.nectarExtractedInBout + self.requiredReduction
        self.totalNectarCollectedOverLife = self.totalNectarCollectedOverLife + self.requiredReduction

        ##print "  --> Extracted " + str(self.requiredReduction) + " unit(s) of nectar"

        if self.remainingNectarCapacity <= 0:
            self.remainingNectarCapacity = self.fixedNectarCapacity          # *** IMPORTANT - this means that the bee's
remainingNectarCapacity should be reset if it is now full
            self.full = 'true'                                  # this means that the action-values will be recalculated on 'empty-of-nectar'
numbers, ready for
                                                # the first decision of the next bout (if any)
    def recalculateEstimatedQuality(self):
        tempStoreNewlyCalculatedQuality = 0.0                   # temporarily stores new calculation output

        # we update the bee's estimated quality of the field in which it foraged using beta to add a measure of caution
```

```
        tempStoreNewlyCalculatedQuality = (self.beta * self.requiredReduction) + ((1 - self.beta) *
self.tempStorePreviousQualityEstimate)
        self.estimatedQualityTable[self.destinationField] = tempStoreNewlyCalculatedQuality

        # if an estimated quality of a field is 0, then the estimated quality is replaced with a VERY small number to
        # avoid action value calculations that attempt to divide by 0
        # ** Check only fields (not nest) for this
        for dylIF in range(1, len(self.estimatedQualityTable)):
            if self.estimatedQualityTable[dylIF] == 0:
                self.estimatedQualityTable[dylIF] = 0.00000000000000000001

        ##print "      --> New Estimated Quality (" + str(self.destinationField) + ") = " +
str(round(self.estimatedQualityTable[self.destinationField], 3))
        ##print ""

    def recalculateActionValues(self):
        tempDivideResult = 0.0                          # temporarily stores calculation output
        newActionValue = 0.0                            # temporarily stores newly calculated action value
        replacementActionValue = 0.0                        # temporarily stores result of applying alpha to new action value

        # for each x in value(x,y)
        for dylIG in range(0, len(self.estimatedQualityTable)):
            # for each y in value(x,y) except y = 0 (the NEST) as 0 is not a valid destination when SELECTING an action-value
            for dylIH in range(1, len(self.estimatedQualityTable)):
                # perform calculations and replace the existing value
                tempDivideResult = self.samplingInterval * (self.remainingNectarCapacity / (self.estimatedQualityTable[dylIH]))
                #if self.beeID == 0:
                    ##print "source = " + str(dylIG)
                    ##print "dest   = " + str(dylIH)
                    ##print "DIST   = " + str(self.distanceMatrix[dylIG][dylIH])
                newActionValue = (self.distanceMatrix[dylIG][dylIH]) + tempDivideResult
                replacementActionValue = (self.dylActionValueTable[dylIG][dylIH]) + (self.alpha * (newActionValue -
(self.dylActionValueTable[dylIG][dylIH])))
                self.dylActionValueTable[dylIG][dylIH] = replacementActionValue

class Main:
    def runMultipleBees(self, nest):
        globalClock = 0                         # tracks the simulated time (in time units) elapsed in the model (NOT REAL TIME)
        injectionInterval = 0                   # interval (in time units) between injections of new bees into the simulation
        maximumInjections = 0                       # the maximum number of new bees that should be injected into the simulation
        timeToCompleteATNEST = 0                        # the time the bee should remain at the nest (ie - the time it should stay in
state AT NEST)
        simCutOffTime = 0                       # the time on the global clock when the simulation will terminate - a value of -1
indicates no cutoff (waits until all bees dead)
        actionValueTrackingOnOff = 0                    # 0 = NO action value tracking, 1 = TURNED ON
        depletionConstant = 0                   # (NOT CURRENTLY USED) stores the amount of full flowers emptied per bee
per flower
        meanUptakeCalcOverTimeOnOff = 0                 # 0 = NO mean uptake calc over time, 1 = TURNED ON
        typesOfPatch = []                       # stores a list of strings that represent the types of patches in the model
        dylReplenishmentInterval = []                   # stores list of values, with each representing the interval between a flower
being emptied and being refilled for the patch type corresponding to the element index
        suddenResourceInjectionOnOff = 0                # 0 = NO sudden resource injection, 1 = TURNED ON
        suddenResourceInjectionAmount = 0                   # the amount of full flowers to inject into the chosen field at the chosen
time if ON
        suddenResourceInjectionTime = 0                     # the time on the global clock when the chosen field should be injected if
turned ON
        suddenResourceInjectionField = 0                # field to receive a sudden injection
        colonyOmniscience = 0                   # 0 = Colony is Normal, 1 = Colony uses Actual Qualities for Estimated Qualities
(including initial estimated qualities, which override the initial estimates specified in the file
        dylPauseToCheckParameters = 0                   # 0 = Don't stop program to check generated replenishments and distance
matrix before adding bees, 1 = TURNED ON
        dylMinimalLog = 0                       # 0 = Generate all graphs and full log, 1 = generate no graphs and minimal log

        distanceType = 0                        # 0 = global equidistance, 1 = neighbourhood equidistance

        nextNewBeeID = 0                        # determines the beeID given to the next new bee
        timeOfNextNewBee = 0                        # the time on the global clock at which the next new bee will be created (if not
at maximum number of bees)
        colony = []                         # list of bees in the form of bee objects
        dylDepletionTally = []                  # stores list with each element representing a list of form [field index, flower index]
indicating that the field-flower combination should be depleted to 0
        allFieldsEmpty = 'false'                    # identifies when no more FULL flowers in the landscape
        beesAlive = 'true'                      # identifies that there are still living bees in the simulation
        cutOff = 'false'                        # identifies when the simulation is to be terminated due to a specified cut off time
        numberOfDeadBees = 0                        # tallies the number of bees that have died
        colonyTraffic = []                      # maintains list of traffic quantities for the entire colony
        meanColonyTraffic = 0.0                     # stores the mean traffic quantity across the colony
        colonyPerformance = []                      # maintains list of performance scores for the entire colony
```

```
        meanColonyPerformance = 0.0                 # stores the mean performance score across the colony
        colonyTotalTraffic = 0.0                    # stores the total number of transitions made by all bees in the colony
        dylTotalForages = []                        # stores list with each element representing the total number of times the field
indicated by the element's index was foraged by bees in the colony

        passLifespan = 0                            # stores lifespan to be used in initialisation of bee
        passSamplingInterval = 0                    # stores sampling interval to be used in initialisation of bee
        passfixedNectarCapacity = 0                 # stores fixed nectar capacity to be used in initialisation of bee
        passEpsilon = 0                             # stores epsilon to be used in initialisation of bee
        passAlpha = 0                               # stores alpha to be used in initialisation of bee
        passBeta = 0                                # stores beta to be used in initialisation of bee
        passPlusRate = 0                            # stores plus rate to be used in initialisation of bee
        passMinusRate = 0                           # stores minus rate to be used in initialisation of bee

        passDistanceMatrix = []                     # stores distance matrix to be used in initialisation of bee in format [[dist(0,0),
dist(0,1)...],[dist(1,0)...]...]
        gridWidth = 0                               # stores the number of cells wide in the grid
        gridHeight = 0                              # stores the number of cells high in the grid
        dcRowOfSource = 0                           # DISTANCE CALCULATION - stores the row in the grid of a source patch
        dcColumnOfSource = 0                        # DISTANCE CALCULATION - stores the column in the grid of a source patch
        dcRowOfDest = 0                             # DISTANCE CALCULATION - stores the row in the grid of a destination patch
        dcColumnOfDest = 0                          # DISTANCE CALCULATION - stores the row in the grid of a destination patch
        tempHorizontal = 0                          # DISTANCE CALCULATION - stores the number of horizontal patch moves
between source and dest
        tempVertical = 0                            # DISTANCE CALCULATION - stores the number of vertical patch moves between
source and dest
        tempCalculatedDistance = 0                  # DISTANCE CALCULATION - stores the calculated distance between source
and dest
        tempPreRoundCalculatedDistance = 0.0        # DISTANCE CALCULATION - stores a calculated diagonal distance as
float before rounding

        dylPassActionValueTable = []                # stores list with each element representing a list of action values for visiting all
fields from the field (or NEST (field 0)) indicated by the element's index
        dylPassEstimatedQualityTable = []           # stores list with each element representing a bee's estimated quality of the
field indicated by the element's index

        originalEstimatedQualityTable = []          # stores the original estimated quality table used when setting up new bees

        dylFull = []                                # stores list with each element representing a temporary storage of the number of FULL
flowers from the resource table file for the field indicated by the element's index
        dylTotal = []                               # stores list with each element representing a temporary storage of the total number of
flowers from the resource table file for the field indicated by the element's index

        meanBoutsCompletedThroughClock = []         # a list of the mean number of bouts completed across the colony, with
each consecutive list element representing each consecutive tick of the global clock
        tempBoutsTotalColony = 0.0                  # temporarily stores the total number of bouts completed across the colony at
any time
        tempMeanBouts = 0.0                         # temporarily stores tempBoutsTotalColony / len(colony)
        dylAllActionValues = []                     # stores list with each element representing a list of elements for action-values
with source field = main list element index, with each nested element being a list of action values over time for the analysed bee for
that valid Value(x,y) defined by the main element index, and the nested element index (format => ([ [ [list of value(0,0) over time],
[list of value(0,1) over time] ...] ...]))
        allActionValuesBee = 0                      # id of analysed bee
        maxList = []                                # stores the list of maximum action values for each x, y combination in Value(x,y)
        maxActionValue = 0.0                        # overall maximum action-value for a bee
        listOfBeePerformanceRecords = []            # holds a list of lists of performance over time for each bee that existed in the
colony
        lengthOfBeePerformanceRecords = []          # holds a list of the length of each list in listOfBeePerformanceRecords
        maxPerformanceInColony = 0.0                # stores the maximum performance score achieved in the colony
        highestAchievablePerformance = 0.0          # stores the calculated highest performance score attainable by a bee in the
model set-up
        dylStartingFull = []                        # stores list with each element representing the number of FULL flowers at the start
of the run of the simulation in the field indicated by the element's index
        dylStartingTotal = []                       # stores list with each element representing the number of total flowers at the start of
the run of the simulation in the field indicated by the element's index
        highestStartingField = 0                    # stores the field with the highest starting quality
        distanceToHighestStartingField = 0.0        # distance from the nest to the field with the highest starting quality
        meanColonyPerformanceOverTime = []          # lists the average performance across the colony over time (1st entry =
mean first bout performance etc)
        smallestCompletedBouts = 0                  # the smallest number of bouts completed by a bee in the colony in a run of the
simulation
        tempTotalPerformanceColony = 0.0            # temporarily stores the sum of performance scores across the colony for a
given number of bouts completed
        tempMeanCalc = 0.0                          # temporarily stores a calculated mean
        tempTimeExisted = 0.0                       # temporarily stores the amount of time a bee has existed
        tempRateOfUptake = 0.0                      # temporarily stores a calculated rate of nectar uptake
        everyUptakeValue = []                       # stores a list of every nectar uptake rate value in the colony
        tempExternalList = []                       # temporarily stores the reconstruction of a list
```

```
    tempLen = 0                                 # temporarily stores the length of a list
    expectedRateNectarUptake = 0.0              # stores the calculated expected rate of nectar uptake (assuming unchanging
resources and random field visitations)
    equalDividedCapacity = 0.0                  # stores (lifespan * fixed nectar capacity) / number of fields in landscape
    capTimesLifespan = 0.0                      # stores (lifespan * fixed nectar capacity)
    dylTimeEstForage = []                       # stores list with each element representing (equalDividedCapacity / quality (x))
where x is the field indicated by the element's index
    equidistantDistance = 0.0                   # stores the equidistant distance between entities in the landscape, taken from
distance (NEST, A)
    equidistantDistanceMultiplied = 0.0         # stores the equidistant distance multiplied up for NEST -> Field and Field ->
NEST for all bouts
    meanOverallUptakeRate = 0.0                 # stores the calculated overall mean nectar uptake rate
    tempSum = 0.0                               # temporarily stores a sum
    dylNextReplenishment = []                   # stores list with each element representing the time (on the global clock) of the
next replenishment of the field indicated by the element's index
    allTrialPerformances = []                   # lists the TRIAL PERFORMANCE SCORE for each bee in the colony
    tempTrialPerformance = 0.0                  # temporarily stores a TRIAL PERFORMANCE SCORE
    colonyTrialPerformance = 0.0                # the mean TRIAL PERFORMANCE SCORE of the colony
    landscapeStandingCropOverTime = []          # stores the total standing crop (number of FULL flowers) in the entire
landscape over time - each consecutive list element represents consecutive ticks of the global clock
    landscapeStandingCropOverTimeAveAcrossFields = []   # stores the average standing crop across the fields in the landscape
over time - each consecutive list element represents consecutive ticks of the global clock
    newFlowersAddedToSystem = 0                 # tallies the number of new flowers added to the simulation during a trial
    tempOldFull = 0                             # temporarily stores the number of FULL flowers in the landscape before the
replenishment
    tempNewFull = 0                             # temporarily stores the number of FULL flowers in the landscape after the
replenishment
    flowersDepletedFromSystem = 0               # tallies the number of FULL flowers sampled by the colony over the trial
    timesWhenBeeSampledEMPTYFlowers = []        # lists the times (on the global clock) when the analysed bee sampled
EMPTY flowers
    tempDummyList = []                          # used to store dummy data

    rankOfTenthPercentileUptake = 0.0           # stores the calculated rank of the 10th Percentile in the Rate of Uptake Data
Set
    rankOfNinetiethPercentileUptake = 0.0       # stores the calculated rank of the 90th Percentile in the Rate of Uptake
Data Set
    tenthPercentileUptake = 0.0                 # the 10th percentile value in the Uptake Data Set
    ninetiethPercentileUptake = 0.0             # the 90th percentile value in the Uptake Data Set
    rankOfTenthPercentileTraffic = 0.0          # stores the calculated rank of the 10th Percentile in the Traffic Data Set
    rankOfNinetiethPercentileTraffic = 0.0      # stores the calculated rank of the 90th Percentile in the Traffic Data Set
    tenthPercentileTraffic = 0.0                # the 10th percentile value in the Traffic Data Set
    ninetiethPercentileTraffic = 0.0            # the 90th percentile value in the Traffic Data Set
    tempIR = 0                                  # temporarily stores the integer part of the calculated rank
    tempFR = 0.0                                # temporarily stores the fraction part of the calculated rank
    tempScoreIR = 0.0                           # temporarily stores the score relating to rank as per IR
    tempScoreIRPlusOne = 0.0                    # temporarily stores the score relating to rank as per IR + 1
    meanUptakeRateAllValues = 0.0               # stores the mean of ALL uptake values from the colony in the simulation
    dylFullOverTime = []                        # stores list with each element representing the number of FULL flowers at each
tick of the global clock in the field indicated by the element's index
    dylTempMaxQuality = 0.0                     # temporarily stores the maximum quality in the landscape
    dylTempMinQuality = 0.0                     # temporarily stores the minimum quality in the landscape
    dylRangeInStandingCropOverTime = []         # stores list with each consecutive element representing the range in
landscape standing crop after each tick of the global clock

    dylNumberOfFieldsInLandscape = 0            # stores the number of fields to be generated in the landscape
    proposedMeanLandscapeQuality = 0.0          # stores the proposed mean landscape quality (about which the field
qualities will be randomly selected (from a uniform distribution))
    rangeInQualities = 0.0                      # stores the range between the highest and lowest field quality.  Specifying this sets
the upper and lower field qualities uniform about the mean (except where < 0 or > 1)
    dylMinimumFieldQualityInLandscape = 0.0     # stores the minimum (starting) field quality to be generated in the
landscape
    dylMaximumFieldQualityInLandscape = 0.0     # stores the maximum (starting) field quality to be generated in the
landscape
    dylFieldCapacity = []                       # stores a list of the total capacity of the patch according to patch type to be
generated in the landscape
    dylDefaultLandscapeSeparation = 0           # stores the default (and equidistant) distance between all entities (fields and
nest) in the landscape
    nominatedBestField = 1                      # stores the randomly nominated field to represent the best field in the landscape
    nominatedWorstField = 1                     # stores the randomly nominated field to represent the worst field in the
landscape
    clashNominations = 1                        # flag to indicate if nominated fields clash

    dylGeneratedLandscape = []                  # stores list in which each element represents a generated field in the landscape,
expressed as a list of form [FULLs at start, Total Flowers at start], with the index of the main list element indicating the field
number
    dylTempGenerateFull = 0                     # temporarily stores a calculated number of FULL flowers to be put into an
element of dylGeneratedLandscape
```

```
    dylTempRandomQuality = 0.0                  # temporarily stores a randomly generated float (min = min field qual, max = max field qual) from a uniform distribution to randomly generate field qualities for the third field (field 2) onwards
    dylTempRawReplenishmentRate = 0.0           # temporarily stores a calculated replenishment rate as a float

    dylNumberOfUniqueFieldsVisited = []         # stores list with each element representing the number of unique fields visited by the bee whose id is indicated by the element's index
    dylMeanNumberOfUniqueFieldsVisited = 0.0    # mean number of unique fields visited across the colony

    transitionMatrix = []                       # stores list with each element containing a sublist, with the 2D array structured : [source][destination] = number of transactions in colony
    totalOfTransitionMatrix = 0                 # stores the total from all cells in the transition matrix (non-percentages)
    runningSumTransitionMatrix = 0              # stores a running total when adding up the cells to calculate totalOfTransitionMatrix

    metaListOfUniquesPerBout = []               # stores a list of listOfUniquesPerBout (one from each forager in the colony)
    LENGTHSmetaListOfUniquesPerBout = []        # stores a list of the lengths of each listOfUniquesPerBout in the metaListOfUniquesPerBout
    lengthOfShortestUniqueList = 0              # stores the length of the shortest listOfUniquesPerBout in the colony

    listOfMeanUniques = []                      # stores list with each element representing the mean unique fields visited across the colony at the time unit defined by the element's index
    listOfMeanPerformances = []                 # stores list with each element representing the mean foraging performance across the colony at the time unit defined by the element's index
    listOfMeanBoutUniques = []                  # stores list with each element representing the mean unique fields visited in the foraging bout indicated by the index across the colony.

    startingFullSorted = []                     # stores a sorted version of dylStartingFull
    fullsOfHighestEmigrationField = 0           # stores the starting FULLs of the highest emigration field
    rankingOfHighestEmigrationField = 0         # stores the ranking of the highest emigration field (in terms of quality)

    unluckiestBee = 0                           # stores the id number of the unluckiest bee in the colony (the one which has the lowest trial performance)
    unluckiestTraffic = 0.0                     # stores the unluckiest bee's mean traffic per bout
    unluckiestPerformance = 0.0                 # stores the unluckiest bee's trial performance score
    unluckiestBoutsCompleted = 0.0              # stores the unluckiest bee's number of bouts completed in trial
    unluckiestUniqueFields = 0.0                # stores the unluckiest bee's number of unique fields visited in trial

    squaredDifferenceTraffic = []               # stores list with each element representing (meanTraf of bee x - mean meanTraf across colony)^2
    squaredDifferencePerformance = []           # stores list with each element representing (perf of bee x - mean perf across colony)^2
    squaredDifferenceBoutsCompleted = []        # stores list with each element representing (bouts of bee x - mean bouts across colony)^2
    squaredDifferenceUniqueFields = []          # stores list with each element representing (uniques of bee x - mean unqiues across colony)^2
    varianceTraffic = 0.0                       # stores variance in traffic (bee-level)
    variancePerformance = 0.0                   # stores variance in performance (bee-level)
    varianceBoutsCompleted = 0.0                # stores variance in bouts completed (bee-level)
    varianceUniqueFields = 0.0                  # stores variance in unique fields (bee-level)
    standardDeviationTraffic = 0.0              # stores standard deviation in traffic (bee-level)
    standardDeviationPerformance = 0.0          # stores standard deviation in performance (bee-level)
    standardDeviationBoutsCompleted = 0.0       # stores standard deviation in bouts completed (bee-level)
    standardDeviationUniqueFields = 0.0         # stores standard deviation in unique fields (bee-level)
    standardErrorTraffic = 0.0                  # stores standard error of traffic (bee-level)
    standardErrorPerformance = 0.0              # stores standard error of performance (bee-level)
    standardErrorBoutsCompleted = 0.0           # stores standard error of bouts completed (bee-level)
    standardErrorUniqueFields = 0.0             # stores standard error of unique fields (bee-level)

    tempGeneSum = 0.0                           # temporary storage parameter
    tempPreviousGeneSum = 0.0                   # temporary storage parameter
    highestEmigrationField = 0                  # stores the field with the highest emigration rate
    geneFlowE = 0.0                             # stores E in gene flow calculation (prob come from GM field)
    geneFlowFF = 1.0                            # stores Fruits Fertilised in gene flow calculation
    geneFlowb = 0.0                             # stores b in gene flow calculation (residence)
    averagePercentStaysNonGM = 0.0              # stores average percentage of stays in fields which are not the GM field
    calculatedGeneFlow = 0.0                    # stores calculated gene flow

    explicitFlowerAllFields = []                # stores a list with each element a list of explicit flower rewards, with outer list element representing the field (0 - the nest - is left empty)
    tempExplicitFull = 0                        # temporarily stores the number of explicit FULL flowers that will need to be set up in a field
    tempExplicitTotal = 0                       # temporarily stores the total number of explicit flowers that will need to be set up in a field
    replenishmentLog = []                       # stores a list with each element a list of form [field, flower, time of next replenishment] - these are placed when a flower is depleted, and removed once the replenishment has occurred
    tempFullReportToFile = 0                    # temporarily stores the number of FULL flowers in a field for quality calculation and subsequent reporting to file
```

```
    tempTotalReportToFile = 0                    # temporarily stores the total number of flowers in a field for quality calculation
and subsequent reporting to file
    tempQualityReportToFile = 0.0                    # temporarily stores tempFullReportToFile / tempTotalReportToFile for
subsquent reporting to file

    accuracyIndicator = 0                    # 0 = spot-on estimate of quality of patch, -1 = estimate is lower than actual, 1 =
estimate is higher than actual

    emulatedReplenishmentInterval = 0                # stores a randomly generated number which represents the remaining time
until replenishment for initially empty flowers

    flowerMatches = 0                    # stores the number of explicit flowers in the depletion tally matching the chosen
flower of the probed bee
    fieldMatches = 0                    # stores the number of fields in the depletion tally matching the chosen field of the
probed bee
    acceptableFlowerChosen = 0                    # 0 = acceptable flower (one not chosen by another bee) not chosen, 1 =
accepatable flower chosen
    chosenFlowerLog = []                    # same as dylDepletionTally, but logs EMPTY flowers as well

    estimatedReplenishmentModel = 0                # 0 = EXPONENTIAL MODEL, 1 = LINEAR MODEL
    gamma = 0.0                    # stores value of gamma (in EXPONENTIAL MODEL, old est * gamma^time
difference; in LINEAR MODEL, old est + gamma for each time step)
    allBeePatchTimeStamps = []                    # stores list of lists, with each sub-list representing seperate bee, and each
element in sub-list representing time stamp of last visit for patch no. = INDEX + 1
    tempTimeDifference = 0                    # temporarily stores difference globalClock - time stamp for a patch for a bee
    capEstQual = 0                    # 0 = don't cap estimated quality of a patch at 1.0, 1 = do.

    totalReadOfPatchTypes = ""                    # string that stores the entire file string of the types of patch file
    patchTypesFileSplitIntoLines = []                # stores list of strings that are separate lines in the types of patch file
    patchTypesFileLinesSplitByComma = []                    # stores list of strings for each line in the types of patch file, split by
comma
    numberOfPatchesOfThisType = []                    # stores list where each element is the number of patches that should be set
up with the patch type corresponding to the element index
    patchTypeNominations = []                    # stores list of lists, with each sub-list a list of patches of type corresponding to
the index of the sub-list in the main list

    trivialDistanceNestSwitch = 0                    # 1 = ON (nest is distance = 1 time unit from all patches), 0 = OFF (nest-field
distances calculated as per grid)
    omega = 0.0                    # stores the probability of a bee that has returned to the nest having its action-values
and estimated qualities reset to defaults (simulating naivety)
    naivetyAssessment = 0.0                    # temporarily stores a randomly generated number to determine (by referring to
omega) if a bee's action-values and estimated qualities should be reset to default

    tempSumPersonalResidence = 0                # temporarily stores a running sum of personal residence
    tempMeanPersonalResidence = 0.0                # temporarily stores a mean personal residence value

    densityPerPatch = []                    # list which stores lists of form [no. of bees in patch 1, no. of bees in patch 2 etc],
with a new sub-list for each time-unit in the simulation
    tempDensityEntry = []                    # temporarily stores a constructed entry for densityPerPatch

    autoAssignPatchTypes = 0                    # 1 = ON (poor patches auto set to EVENs, good patches auto set to ODDs), 0 =
OFF
    autoAssignCounter = 0                    # counter to keep track of next automatically assigned patch

    allResidenceLog = []                    # stores the mean residence across ALL patches for each bee in colony, with each
element representing separate bee
    specificResidenceLog = []                    # stores a list of lists - each sublist represents a separate patch type, and within :
the mean residence in patches for each bee in colony, with each element representing separate bee
    eachResidenceLog = []                    # stores a list of lists - each sublist represents a separate patch, and within : the
mean residence in patches for each bee in colony, with each element representing separate bee
    ALLVisitsLog = []                    # stores the total number of patch visits to ALL patches from each bee in the colony
    POORVisitsLog = []                    # stores the total number of patch visits to POOR patches from each bee in the
colony
    GOODVisitsLog = []                    # stores the total number of patch visits to GOOD patches from each bee in the
colony
    EACHVisitsLog = []                    # stores list of lists, with each sub-list representing a patch and within the total
number of patch visits to that patch from each bee in the colony

    # temporary storage parameters
    tempString = ''
    tempFloat = 0.0
    tempRead = ''
    tempDisFirst = ''
    tempDis = ''
    tempRewFirst = ''
    tempRewSecond = ''
    tempRewThird = ''
    tempRew = ''
```

```
tempRewIntFirst = 0
tempRewInt = 0
tempRewFloatFirst = 0.0
tempRewFloat = 0.0
tempRewRunningTally = 0.0
tempInt = 0
tempTransMatValue = 0
tempMeanUniques = 0.0
tempUniquesSum = 0.0
tempMeanPerformances = 0.0
tempPerformancesSum = 0.0
tempMeanBoutUniques = 0.0
tempBoutUniquesSum = 0.0

# GENERATION OF THE LANDSCAPE
# ---------------------------

# open types of patch file, and read and store the types of patch to be used
typesOfPatchFile = open('typesOfPatchFile.txt','r')

totalReadOfPatchTypes = typesOfPatchFile.read()                    # read the entire file as a single string
patchTypesFileSplitIntoLines = totalReadOfPatchTypes.splitlines()   # split the string into lines

for iz in range(0, len(patchTypesFileSplitIntoLines)):
   patchTypesFileLinesSplitByComma.append(patchTypesFileSplitIntoLines[iz])

typesOfPatchFile.close()

for ia in range(0, len(patchTypesFileSplitIntoLines)):
   tempString = patchTypesFileSplitIntoLines[ia].split(',')
   typesOfPatch.append(tempString[0])
   dylReplenishmentInterval.append(int(tempString[1]))
   numberOfPatchesOfThisType.append(int(tempString[2]))
   # store total number of flowers in patches of this type
   dylFieldCapacity.append(int(tempString[3]))

# open dynamic landscape parameters file to store the parameters used to generate the landscape
dynamicLandscapeParametersFile = open('dynamicLandscapeParametersFile.txt','r')

# read in and store the parameters
tempRead = dynamicLandscapeParametersFile.readline()
dylNumberOfFieldsInLandscape = int(dynamicLandscapeParametersFile.readline())

tempRead = dynamicLandscapeParametersFile.readline()
proposedMeanLandscapeQuality = float(dynamicLandscapeParametersFile.readline())

tempRead = dynamicLandscapeParametersFile.readline()
rangeInQualities = float(dynamicLandscapeParametersFile.readline())

tempRead = dynamicLandscapeParametersFile.readline()
dylDefaultLandscapeSeparation = int(dynamicLandscapeParametersFile.readline())

# read in whether patches should be auto assigned
modelTypeFile = open('modelTypeFile.txt','r')

modelTypeFile.readline()
modelTypeFile.readline()
modelTypeFile.readline()
modelTypeFile.readline()
modelTypeFile.readline()
modelTypeFile.readline()
modelTypeFile.readline()
autoAssignPatchTypes = int(modelTypeFile.readline())

modelTypeFile.close()

tempInt = 0

# get the user to input the fields which will be of each types as specified in the types of patch file
# (or do automatically if feature switched on)
for ia in range(0, len(numberOfPatchesOfThisType)):
   patchTypeNominations.append([])                    # start a new sub-list for this patch type

   # reset counter according to current patch type
   if typesOfPatch[ia] == "Poor":
      autoAssignCounter = 2
   if typesOfPatch[ia] == "Good":
      autoAssignCounter = 1
```

```
            for ib in range (0, numberOfPatchesOfThisType[ia]):
                if autoAssignPatchTypes == 0:
                    tempInt = int(raw_input(("Please nominate a patch to be " + str(typesOfPatch[ia]) + " : ")))          # get the user to
nominate a patch to be of this type
                else:
                    tempInt = int(autoAssignCounter)
                    autoAssignCounter = autoAssignCounter + 2

                patchTypeNominations[ia].append(tempInt)              # append the nominated patch to the list for this type

        tempInt = 0

        # set up tempDensityEntry with dummy entries according to the number of patches in the landscape
        for ia in range(0, dylNumberOfFieldsInLandscape):
            tempDensityEntry.append(0)

        # calculate the lower field quality and store it
        dylMinimumFieldQualityInLandscape = proposedMeanLandscapeQuality - (rangeInQualities / 2.0)

        # if minimum quality is less than 0, set to 0.0
        if dylMinimumFieldQualityInLandscape < 0:
            dylMinimumFieldQualityInLandscape = 0.0

        # calculate the upper field quality and store it
        dylMaximumFieldQualityInLandscape = proposedMeanLandscapeQuality + (rangeInQualities / 2.0)

        # if maximum quality is more than 1, set to 1.0
        if dylMaximumFieldQualityInLandscape > 1:
            dylMaximumFieldQualityInLandscape = 1.0

        # INITIALISATION of certain lists
        for dylIZ in range(0, (dylNumberOfFieldsInLandscape + 1)):
            dylDepletionTally.append([])
            dylTotalForages.append(0)
            dylFull.append(0)
            dylTotal.append(0)
            dylTimeEstForage.append(0.0)
            dylNextReplenishment.append(0)
            dylFullOverTime.append([])
            chosenFlowerLog.append([])

        # initialisation of the transitionMatrix
        for ia in range(0, (dylNumberOfFieldsInLandscape + 1)):
            transitionMatrix.append([])

        for ib in range(0, (dylNumberOfFieldsInLandscape + 1)):
            for ic in range(0, (dylNumberOfFieldsInLandscape + 1)):
                transitionMatrix[ib].append(0)

        # open file with user defined global parameters
        globalParametersFile = open ('globalParametersFile.txt', 'r')

        # read in global parameters from the file
        tempRead = globalParametersFile.readline()          # read label line first in each pair to skip
        injectionInterval = int(globalParametersFile.readline())

        tempRead = globalParametersFile.readline()
        maximumInjections = int(globalParametersFile.readline())

        tempRead = globalParametersFile.readline()
        timeToCompleteATNEST = int(globalParametersFile.readline())

        tempRead = globalParametersFile.readline()
        simCutOffTime = int(globalParametersFile.readline())

        tempRead = globalParametersFile.readline()
        actionValueTrackingOnOff = int(globalParametersFile.readline())

        tempRead = globalParametersFile.readline()
        allActionValuesBee = int(globalParametersFile.readline())

        tempRead = globalParametersFile.readline()
        depletionConstant = int(globalParametersFile.readline())

        tempRead = globalParametersFile.readline()
        meanUptakeCalcOverTimeOnOff = int(globalParametersFile.readline())
```

```
tempRead = globalParametersFile.readline()
suddenResourceInjectionOnOff = int(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
suddenResourceInjectionAmount = int(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
suddenResourceInjectionTime = int(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
suddenResourceInjectionField = int(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
colonyOmniscience = int(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
dylPauseToCheckParameters = int(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
dylMinimalLog = int(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
gamma = float(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
trivialDistanceNestSwitch = int(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
omega = float(globalParametersFile.readline())

tempRead = globalParametersFile.readline()
useDepletionAndReplenishment = int(globalParametersFile.readline())

tempRead = ''
globalParametersFile.close()

# initialise the allBeePatchTimeStamps list
for ia in range(0, maximumInjections):
    allBeePatchTimeStamps.append([])
    # for all fields for this bee, set time stamp to 0
    for ib in range(0, dylNumberOfFieldsInLandscape):
        allBeePatchTimeStamps[ia].append(0)

# generate the landscape from the parameters
# check to see if patches were set up with precision
precisePatchFile = open('precisePatchFile.txt','r')

precisePatchFile.readline()
tempCheckOnThePP = int(precisePatchFile.readline())

precisePatchFile.close()

if tempCheckOnThePP == 1:
    # generate according to precise patch file
    precisePatchFile = open('precisePatchFile.txt','r')

    # skip header
    precisePatchFile.readline()
    precisePatchFile.readline()

    # set up nest
    dylGeneratedLandscape.append([0,0])

    for i in range(1, (dylNumberOfFieldsInLandscape + 1)):
        precisePatchFile.readline()
        tempIntPP1 = int(precisePatchFile.readline())
        tempIntPP2 = int(precisePatchFile.readline())

        dylGeneratedLandscape.append([tempIntPP1, tempIntPP2])

    precisePatchFile.close()
else:
    # repeat the following for every field needed to be generated
    for dylIA in range(0, (dylNumberOfFieldsInLandscape + 1)):
        # if this is first entity added, then set up the nest as field 0
        if len(dylGeneratedLandscape) == 0:
            dylGeneratedLandscape.append([0,0])
        else:
```

```
            # if the nest has been added
            dylTempRandomQuality = random.uniform(dylMinimumFieldQualityInLandscape,
dylMaximumFieldQualityInLandscape)    # randomly generate a field quality
            # find the patch type of this patch and use field capacity accordingly
            for i in range(0, len(patchTypeNominations)):
               for j in range(0, len(patchTypeNominations[i])):
                  if dylIA == patchTypeNominations[i][j]:
                     dylTempGenerateFull = int(dylTempRandomQuality * dylFieldCapacity[i])          # calculate number of
FULL flowers based on RANDOMLY GENERATED quality (cast as integer in case result is float)
                     dylGeneratedLandscape.append([dylTempGenerateFull, dylFieldCapacity[i]])       # add the [FULL, total]
pair to the list, with the index now representing the field number

         # now select two fields at random (but not field 0 as that is the nest) and give one the maximum field quality,
         # and one the minimum field quality
         nominatedBestField = random.randint(1, dylNumberOfFieldsInLandscape)          # randomly select best field
         while (clashNominations == 1):                                                # randomly select worst field until no clash with best
field
            nominatedWorstField = random.randint(1, dylNumberOfFieldsInLandscape)
            if nominatedWorstField != nominatedBestField:
               clashNominations = 0

         # set up best field
         # find the patch type of this patch and use field capacity accordingly
         for i in range(0, len(patchTypeNominations)):
            for j in range(0, len(patchTypeNominations[i])):
               if nominatedBestField == patchTypeNominations[i][j]:
                  dylTempGenerateFull = int(dylMaximumFieldQualityInLandscape * dylFieldCapacity[i])      # calculate number of
FULL flowers based on MAXIMUM quality (cast as integer in case result is float)
                  dylGeneratedLandscape[nominatedBestField] = [dylTempGenerateFull, dylFieldCapacity[i]]

         # set up worst field
         for i in range(0, len(patchTypeNominations)):
            for j in range(0, len(patchTypeNominations[i])):
               if nominatedWorstField == patchTypeNominations[i][j]:
                  dylTempGenerateFull = int(dylMinimumFieldQualityInLandscape * dylFieldCapacity[i])        # calculate number of
FULL flowers based on MINIMUM quality (cast as integer in case result is float)
                  dylGeneratedLandscape[nominatedWorstField] = [dylTempGenerateFull, dylFieldCapacity[i]]

      # write the generated landscape to the resource table file
      resourceTable = open ('resourceTable.txt','w')

      for dylIB in range(1, (dylNumberOfFieldsInLandscape + 1)):
         resourceTable.write(("Field " + str(dylIB)))
         resourceTable.write("\n")
         resourceTable.write(str(dylGeneratedLandscape[dylIB][0]))
         resourceTable.write("\n")
         resourceTable.write(str(dylGeneratedLandscape[dylIB][1]))
         resourceTable.write("\n")

      resourceTable.close()

      # write the generated landscape to the initialResourceTable file to record the initial resources
      initialResourceTable = open('initialResourceTable.txt','w')

      for ix in range(1, (dylNumberOfFieldsInLandscape + 1)):
         initialResourceTable.write(("Field " + str(ix)))
         initialResourceTable.write("\n")
         initialResourceTable.write(str(dylGeneratedLandscape[ix][0]))
         initialResourceTable.write("\n")
         initialResourceTable.write(str(dylGeneratedLandscape[ix][1]))
         initialResourceTable.write("\n")

      initialResourceTable.close()

      # initialise explicitFlowerAllFields list
      for iz in range(0, (dylNumberOfFieldsInLandscape + 1)):
         explicitFlowerAllFields.append([])

      # set up array of flowers for each field (not the nest), with each element representing the reward of the flower indicated by the
element index
      for ia in range(1, (dylNumberOfFieldsInLandscape + 1)):
         tempExplicitFull = dylGeneratedLandscape[ia][0]          # grab the number of FULLs that will need to be explicitly
represented in the field
         tempExplicitTotal = dylGeneratedLandscape[ia][1]         # grab the total number of flowers that will need to be
explicitly represented in the field

         # set all flowers initially to EMPTY
         for ib in range(0, tempExplicitTotal):
```

```
            explicitFlowerAllFields[ia].append(0)

        # set x flowers to FULL where x = tempExplicitFull
        for ic in range(0, tempExplicitFull):
            explicitFlowerAllFields[ia][ic] = 1

        # for all initially empty flowers, append records to the replenishment log to mark them for future replenishment
        # (randomly generate the next replenishment for each initial empty)
        for ix in range(0, tempExplicitTotal):
            if explicitFlowerAllFields[ia][ix] == 0:
                # find the patch type of the field in question
                for ij in range(0, len(patchTypeNominations)):
                    for ik in range(0, len(patchTypeNominations[ij])):
                        if patchTypeNominations[ij][ik] == ia:          # if field matches
                            tempInt = ij                                # note the outer index as this corresponds with patch type

                emulatedReplenishmentInterval = random.randint(1, dylReplenishmentInterval[tempInt])
                if useDepletionAndReplenishment == 1:
                    replenishmentLog.append([ia, ix, int(globalClock + emulatedReplenishmentInterval)])

                tempInt = 0

    # add dummy nest totals
    dylStartingFull.append(0)
    dylStartingTotal.append(0)

    # store the starting resources for future reference
    for dylIC in range(1, (dylNumberOfFieldsInLandscape + 1)):
        dylStartingFull.append(dylGeneratedLandscape[dylIC][0])

    for dylID in range(1, (dylNumberOfFieldsInLandscape + 1)):
        dylStartingTotal.append(dylGeneratedLandscape[dylID][1])

    # open file with user defined bee parameters
    beeParametersFile = open ('beeParametersFile.txt', 'r')

    # read in parameters for the bee from the file
    tempRead = beeParametersFile.readline()
    passLifespan = int(beeParametersFile.readline())

    tempRead = beeParametersFile.readline()
    passSamplingInterval = int(beeParametersFile.readline())

    tempRead = beeParametersFile.readline()
    passfixedNectarCapacity = float(beeParametersFile.readline())

    tempRead = beeParametersFile.readline()
    passEpsilon = float(beeParametersFile.readline())

    tempRead = beeParametersFile.readline()
    passAlpha = float(beeParametersFile.readline())

    tempRead = beeParametersFile.readline()
    passBeta = float(beeParametersFile.readline())

    tempRead = beeParametersFile.readline()
    passPlusRate = float(beeParametersFile.readline())

    tempRead = beeParametersFile.readline()
    passMinusRate = float(beeParametersFile.readline())

    tempRead = ''
    beeParametersFile.close()

    # set up initial inter-field / nest-field distances - populate distance matrix
    modelTypeFile = open('modelTypeFile.txt','r')

    modelTypeFile.readline()
    distanceType = int(modelTypeFile.readline())

    modelTypeFile.readline()
    estimatedReplenishmentModel = int(modelTypeFile.readline())

    modelTypeFile.readline()
    capEstQual = int(modelTypeFile.readline())

    modelTypeFile.readline()
    autoAssignPatchTypes = int(modelTypeFile.readline())
```

```
modelTypeFile.close()

# check to see if patch coordinates were explicitly defined
patchCoordinatesFile = open('patchCoordinatesFile.txt','r')

patchCoordinatesFile.readline()
tempCheckOnTheCOORDS = int(patchCoordinatesFile.readline())

patchCoordinatesFile.close()

if tempCheckOnTheCOORDS == 1:
   patchCoordinatesList1 = []
   patchCoordinatesList2 = []
   patchCoordinatesList3 = []
   tempCoordsHolder = []

   patchCoordinatesFile = open('patchCoordinatesFile.txt','r')

   patchCoordinatesFileString = patchCoordinatesFile.read()

   patchCoordinatesList1 = patchCoordinatesFileString.splitlines()

   for i in range(2, len(patchCoordinatesList1)):
      if patchCoordinatesList1[i].startswith("Patch"):
         print "rejecting header..."
      else:
         patchCoordinatesList2.append(patchCoordinatesList1[i])

   for i in range(0, len(patchCoordinatesList2)):
      tempCoordsString = patchCoordinatesList2[i]

      tempCoordsHolder = tempCoordsString.split(",")

      patchCoordinatesList3.append([int(tempCoordsHolder[0]), int(tempCoordsHolder[1])])

   patchCoordinatesFile.close()

   xDifference = 0.0
   yDifference = 0.0
   hypotenuse = 0.0
   hypotenuseINT = 0

   # find out the coordinates of the nest (to be used if trivial distance nest switched off)
   nestX = 0
   nestY = 0

   nestParametersFile = open('nestParametersFile.txt','r')

   nestParametersFile.readline()
   nestX = int(nestParametersFile.readline())
   nestParametersFile.readline()
   nestY = int(nestParametersFile.readline())

   nestParametersFile.close()

   # set up nest-field distances
   passDistanceMatrix.append([])
   for dylIG in range(0,(dylNumberOfFieldsInLandscape + 1)):
      if dylIG == 0:
         passDistanceMatrix[0].append(0)
      else:
         # if trivial distance nest switch off then calculate as normal (nest assumed to be at (0,0)), else set distance to 1
         if trivialDistanceNestSwitch == 0:
            xDifference = math.sqrt(pow((patchCoordinatesList3[(dylIG - 1)][0] - nestX),2))
            yDifference = math.sqrt(pow((patchCoordinatesList3[(dylIG - 1)][1] - nestY),2))
            hypotenuse = math.sqrt((pow(xDifference,2) + pow(yDifference,2)))

            # if calculated distance (in time units) is not an exact integer, then add 1 time unit to the distance, as time units are
atomic
            # and need to indicate that will take extra time unit to travel the remainder above the exact integer
            if (hypotenuse / int(hypotenuse)) > 1.0:
               hypotenuse = hypotenuse + 1

            hypotenuseINT = int(hypotenuse)

            passDistanceMatrix[0].append(hypotenuseINT)
         else:
```

```
                passDistanceMatrix[0].append(1)

            # set up field-field and field-nest distances
            for dylIH in range(1,(dylNumberOfFieldsInLandscape + 1)):
               passDistanceMatrix.append([])
               for dylII in range(0,(dylNumberOfFieldsInLandscape + 1)):
                  if dylII == dylIH:
                     passDistanceMatrix[dylIH].append(0)
                  else:
                     # if destination = NEST
                     if dylII == 0:
                        # if trivial distance nest switch off then calculate as normal (nest assumed to be at (0,0)), else set distance to 0
                        if trivialDistanceNestSwitch == 0:
                           xDifference = math.sqrt(pow((patchCoordinatesList3[(dylIH - 1)][0] - nestX),2))
                           yDifference = math.sqrt(pow((patchCoordinatesList3[(dylIH - 1)][1] - nestY),2))
                           hypotenuse = math.sqrt((pow(xDifference,2) + pow(yDifference,2)))

                           # if calculated distance (in time units) is not an exact integer, then add 1 time unit to the distance, as time units
are atomic
                           # and need to indicate that will take extra time unit to travel the remainder above the exact integer
                           if (hypotenuse / int(hypotenuse)) > 1.0:
                              hypotenuse = hypotenuse + 1

                           hypotenuseINT = int(hypotenuse)

                           passDistanceMatrix[dylIH].append(hypotenuseINT)
                        else:
                           passDistanceMatrix[dylIH].append(1)
                     else:
                        # if destination not the nest then calculate as normal
                        xDifference = math.sqrt(pow((patchCoordinatesList3[(dylIH - 1)][0] - patchCoordinatesList3[(dylII - 1)][0]),2))
                        yDifference = math.sqrt(pow((patchCoordinatesList3[(dylIH - 1)][1] - patchCoordinatesList3[(dylII - 1)][1]),2))
                        hypotenuse = math.sqrt((pow(xDifference,2) + pow(yDifference,2)))

                        # if calculated distance (in time units) is not an exact integer, then add 1 time unit to the distance, as time units are
atomic
                        # and need to indicate that will take extra time unit to travel the remainder above the exact integer
                        if (hypotenuse / int(hypotenuse)) > 1.0:
                           hypotenuse = hypotenuse + 1

                        hypotenuseINT = int(hypotenuse)

                        passDistanceMatrix[dylIH].append(hypotenuseINT)

            # write distance matrix to file
            distanceMatrixFile = open('distanceMatrixFile.txt', 'w')

            for dylIJ in range(0,(dylNumberOfFieldsInLandscape + 1)):
               for dylIK in range(0,(dylNumberOfFieldsInLandscape + 1)):
                  if dylIJ == 0:
                     distanceMatrixFile.write("NEST")
                     distanceMatrixFile.write("\n")
                  else:
                     distanceMatrixFile.write(("Field " + str(dylIJ)))
                     distanceMatrixFile.write("\n")

                  if dylIK == 0:
                     distanceMatrixFile.write("NEST")
                     distanceMatrixFile.write("\n")
                  else:
                     distanceMatrixFile.write(("Field " + str(dylIK)))
                     distanceMatrixFile.write("\n")

                  distanceMatrixFile.write(str(passDistanceMatrix[dylIJ][dylIK]))
                  distanceMatrixFile.write("\n")

            distanceMatrixFile.close()
         else:
            # if using global equidistance
            if distanceType == 0:
               # set up nest-field distances
               passDistanceMatrix.append([])
               for dylIG in range(0,(dylNumberOfFieldsInLandscape + 1)):
                  if dylIG == 0:
                     passDistanceMatrix[0].append(0)
                  else:
                     # if trivial distance nest switch off then calculate as normal, else set distance to 1
                     if trivialDistanceNestSwitch == 0:
```

```
            passDistanceMatrix[0].append(dylDefaultLandscapeSeparation)
         else:
            passDistanceMatrix[0].append(1)

   # set up field-field and field-nest distances
   for dylIH in range(1,(dylNumberOfFieldsInLandscape + 1)):
      passDistanceMatrix.append([])
      for dylII in range(0,(dylNumberOfFieldsInLandscape + 1)):
         if dylII == dylIH:
            passDistanceMatrix[dylIH].append(0)
         else:
            # if destination = NEST
            if dylII == 0:
               # if trivial distance nest switch off then calculate as normal, else set distance to 0
               if trivialDistanceNestSwitch == 0:
                  passDistanceMatrix[dylIH].append(dylDefaultLandscapeSeparation)
               else:
                  passDistanceMatrix[dylIH].append(1)
            else:
               # if destination not the nest then calculate as normal
               passDistanceMatrix[dylIH].append(dylDefaultLandscapeSeparation)

   # write distance matrix to file
   distanceMatrixFile = open('distanceMatrixFile.txt', 'w')

   for dylIJ in range(0,(dylNumberOfFieldsInLandscape + 1)):
      for dylIK in range(0,(dylNumberOfFieldsInLandscape + 1)):
         if dylIJ == 0:
            distanceMatrixFile.write("NEST")
            distanceMatrixFile.write("\n")
         else:
            distanceMatrixFile.write(("Field " + str(dylIJ)))
            distanceMatrixFile.write("\n")

         if dylIK == 0:
            distanceMatrixFile.write("NEST")
            distanceMatrixFile.write("\n")
         else:
            distanceMatrixFile.write(("Field " + str(dylIK)))
            distanceMatrixFile.write("\n")

         distanceMatrixFile.write(str(passDistanceMatrix[dylIJ][dylIK]))
         distanceMatrixFile.write("\n")

   distanceMatrixFile.close()

# if using neighbourhood equidistance
if distanceType == 1:
   # read in grid width and hieght from file and update parameters accordingly
   gridParametersFile = open('gridParametersFile.txt','r')

   gridParametersFile.readline()
   gridParametersFile.readline()

   gridParametersFile.readline()
   gridParametersFile.readline()

   gridParametersFile.readline()
   gridWidth = int(gridParametersFile.readline())

   gridParametersFile.readline()
   gridHeight = int(gridParametersFile.readline())

   gridParametersFile.close()

   # set up nest-field distances
   # start a new list for source=nest
   passDistanceMatrix.append([])

   # nest row and column
   dcRowOfSource = 1
   dcColumnOfSource = 0

   # for each destination patch
   for ia in range(0,(dylNumberOfFieldsInLandscape + 1)):
      if ia == 0:                                    # if destination = nest, set destination nest row and column
         dcRowOfDest = 1
         dcColumnOfDest = 0
```

```
            else:                                   # otherwise
               dcRowOfDest = int((ia / gridWidth)) + 1          # calculate row and column of destination
               dcColumnOfDest = int((ia % gridWidth))

               # if dest patch is at the right-hand end of the row, calculation of destination row and column is slightly different
               if (ia % gridWidth) == 0:
                  dcRowOfDest = int(ia / gridWidth)
                  dcColumnOfDest = gridWidth

            # work out number of horizontal and vertical patch moves between source and destination
            tempHorizontal = int(math.sqrt(math.pow((dcColumnOfDest - dcColumnOfSource), 2)))
            tempVertical = int(math.sqrt(math.pow((dcRowOfDest - dcRowOfSource), 2)))

            # work out distance from horizontal and vertical moves
            # if the move is a diagonal move
            if tempHorizontal > 0 and tempVertical > 0:
               # calculate the diagonal using Pythagoras' theorem
               tempPreRoundCalculatedDistance = math.sqrt((math.pow((tempHorizontal * dylDefaultLandscapeSeparation), 2) +
math.pow((tempVertical * dylDefaultLandscapeSeparation), 2)))

               # if calculated distance (in time units) is not an exact integer, then add 1 time unit to the distance, as time units are
atomic
               # and need to indicate that will take extra time unit to travel the remainder above the exact integer
               if (tempPreRoundCalculatedDistance / int(tempPreRoundCalculatedDistance)) > 1.0:
                  tempPreRoundCalculatedDistance = tempPreRoundCalculatedDistance + 1

               # cast the distance as an integer and copy to the correct variable
               tempCalculatedDistance = int(tempPreRoundCalculatedDistance)
            else:
               # if distance not a diagonal, simply work out number horizontal-only or vertical-only distance
               tempCalculatedDistance = int(dylDefaultLandscapeSeparation * tempHorizontal) +
int(dylDefaultLandscapeSeparation * tempVertical)

            # if zero distance nest switch is off then calculate as normal, else set distance to 0
            if trivialDistanceNestSwitch == 0:
               passDistanceMatrix[0].append(tempCalculatedDistance)       # append to the distance matrix
            else:
               passDistanceMatrix[0].append(1)

         # set up field-field distances
         # for each source patch (excluding the nest)
         for ib in range(1,(dylNumberOfFieldsInLandscape + 1)):
            # start a new list for source=ib
            passDistanceMatrix.append([])

            # work out row and column of source patch
            dcRowOfSource = int((ib / gridWidth)) + 1
            dcColumnOfSource = int((ib % gridWidth))

            # if source patch is at the right-hand end of the row, calculation of source row and column is slightly different
            if (ib % gridWidth) == 0:
               dcRowOfSource = int(ib / gridWidth)
               dcColumnOfSource = gridWidth

            # for each destination patch (INCLUDING the nest)
            for ic in range(0,(dylNumberOfFieldsInLandscape + 1)):
               if ic == 0:                          # if destination=nest, set destination nest row and column
                  dcRowOfDest = 1
                  dcColumnOfDest = 0
               else:                                # otherwise
                  dcRowOfDest = int((ic / gridWidth)) + 1          # calculate row and column of destination
                  dcColumnOfDest = int((ic % gridWidth))

                  # if dest patch is at the right-hand end of the row, calculation of destination row and column is slightly different
                  if (ic % gridWidth) == 0:
                     dcRowOfDest = int(ic / gridWidth)
                     dcColumnOfDest = gridWidth

               # work out number of horizontal and vertical patch moves between source and destination
               tempHorizontal = int(math.sqrt(math.pow((dcColumnOfDest - dcColumnOfSource), 2)))
               tempVertical = int(math.sqrt(math.pow((dcRowOfDest - dcRowOfSource), 2)))

               # work out distance from horizontal and vertical moves
               # if the move is a diagonal move
               if tempHorizontal > 0 and tempVertical > 0:
                  # calculate the diagonal using Pythagoras' theorem
                  tempPreRoundCalculatedDistance = math.sqrt((math.pow((tempHorizontal * dylDefaultLandscapeSeparation), 2)
+ math.pow((tempVertical * dylDefaultLandscapeSeparation), 2)))
```

```
                    # if calculated distance (in time units) is not an exact integer, then add 1 time unit to the distance, as time units are
atomic
                    # and need to indicate that will take extra time unit to travel the remainder above the exact integer
                    if (tempPreRoundCalculatedDistance / int(tempPreRoundCalculatedDistance)) > 1.0:
                        tempPreRoundCalculatedDistance = tempPreRoundCalculatedDistance + 1

                    # cast the distance as an integer and copy to the correct variable
                    tempCalculatedDistance = int(tempPreRoundCalculatedDistance)
                else:
                    # if distance not a diagonal, simply work out number horizontal-only or vertical-only distance
                    tempCalculatedDistance = int(dylDefaultLandscapeSeparation * tempHorizontal) +
int(dylDefaultLandscapeSeparation * tempVertical)

                # if destination is the NEST
                if ic == 0:
                    # if trivial distance nest switch is off then calculate as normal, else set distance to 0
                    if trivialDistanceNestSwitch == 0:
                        passDistanceMatrix[ib].append(tempCalculatedDistance)          # append to the distance matrix
                    else:
                        passDistanceMatrix[ib].append(1)
                else:
                    # if destination is not the nest, then calculate as normal
                    passDistanceMatrix[ib].append(tempCalculatedDistance)          # append to the distance matrix

            # write distance matrix to file
            distanceMatrixFile = open('distanceMatrixFile.txt', 'w')

            for dylIJ in range(0,(dylNumberOfFieldsInLandscape + 1)):
                for dylIK in range(0,(dylNumberOfFieldsInLandscape + 1)):
                    if dylIJ == 0:
                        distanceMatrixFile.write("NEST")
                        distanceMatrixFile.write("\n")
                    else:
                        distanceMatrixFile.write(("Field " + str(dylIJ)))
                        distanceMatrixFile.write("\n")

                    if dylIK == 0:
                        distanceMatrixFile.write("NEST")
                        distanceMatrixFile.write("\n")
                    else:
                        distanceMatrixFile.write(("Field " + str(dylIK)))
                        distanceMatrixFile.write("\n")

                    distanceMatrixFile.write(str(passDistanceMatrix[dylIJ][dylIK]))
                    distanceMatrixFile.write("\n")

            distanceMatrixFile.close()

        # PAUSE TO CHECK PARAMETERS if facility turned ON
        # AND if a pause is made, then the replenishment and distance files are read back in (to capture potential changes)
        if dylPauseToCheckParameters == 1:
            raw_input("Check DISTANCE MATRIX.  Then PRESS ENTER")

            # read distances back in and update the distance matrix
            distanceMatrixFile = open('distanceMatrixFile.txt', 'r')

            for dylIM in range(0, (dylNumberOfFieldsInLandscape + 1)):
                for dylIN in range(0, (dylNumberOfFieldsInLandscape + 1)):
                    tempRead = distanceMatrixFile.readline()
                    tempRead = distanceMatrixFile.readline()
                    passDistanceMatrix[dylIM][dylIN] = int(distanceMatrixFile.readline())

            distanceMatrixFile.close()

        # Initial action values are now populated as the distance values (in time units) specified between fields in the distance matrix.
        for dylIO in range(0, (dylNumberOfFieldsInLandscape + 1)):
            dylPassActionValueTable.append([])
            for dylIP in range(0, (dylNumberOfFieldsInLandscape + 1)):
                dylPassActionValueTable[dylIO].append(passDistanceMatrix[dylIO][dylIP])

        # if colony is omniscient, then set initial estimated qualities as the actual field qualities
        # if colony is normal, then set initial estimated qualities as the MEAN of the actual field qualities
        if colonyOmniscience == 1:
            resourceTable = open ('resourceTable.txt','r')

            dylPassEstimatedQualityTable.append(0.0)        # nest DUMMY estimated quality
            for dylIQ in range(1, (dylNumberOfFieldsInLandscape + 1)):
```

```
            tempRead = resourceTable.readline()
            tempRewIntFirst = int(resourceTable.readline())
            tempRewInt = int(resourceTable.readline())
            dylPassEstimatedQualityTable.append((float(tempRewIntFirst) / float(tempRewInt)))

         resourceTable.close()
      else:
         tempRewRunningTally = 0.0

         resourceTable = open('resourceTable.txt','r')

         for dylIR in range(1, (dylNumberOfFieldsInLandscape + 1)):
            tempRead = resourceTable.readline()
            tempRewIntFirst = int(resourceTable.readline())
            tempRewInt = int(resourceTable.readline())
            tempRewFloatFirst = float(tempRewIntFirst) / float(tempRewInt)
            tempRewRunningTally = float(tempRewRunningTally) + float(tempRewFloatFirst)

         resourceTable.close()

         tempRewFloat = float(tempRewRunningTally) / float(dylNumberOfFieldsInLandscape)

         dylPassEstimatedQualityTable.append(0.0)       # nest DUMMY estimated quality
         for dylIS in range(1, (dylNumberOfFieldsInLandscape + 1)):
            dylPassEstimatedQualityTable.append(tempRewFloat)

      # store the original estimated quality table
      for i in range(0, len(dylPassEstimatedQualityTable)):
         originalEstimatedQualityTable.append(float(dylPassEstimatedQualityTable[i]))

      passNestRateState = nest.getNestRateState()                  # CURRENTLY UNUSED - check state of nest nectar build up

      bee = Bee()                                          # creates the first bee of the simulation (required)

      # set up the bee with the initial parameters
      bee.initialSetup(passLifespan, passSamplingInterval, passfixedNectarCapacity, passEpsilon, passAlpha, passBeta,
passPlusRate, passMinusRate, dylPassActionValueTable, passDistanceMatrix, dylPassEstimatedQualityTable, passNestRateState)
      bee.beeID = nextNewBeeID
      bee.beeCreationTime = globalClock
      nextNewBeeID = nextNewBeeID + 1                          # next ID will be 1 higher
      bee.remainingNectarCapacity = bee.fixedNectarCapacity
      bee.state = "AT NEST"
      bee.completionTimeOfState = globalClock + timeToCompleteATNEST

      # append initial estimated qualities to history for each field
      for dylIT in range(1, (dylNumberOfFieldsInLandscape + 1)):
         bee.dylEstimatedQualityHistory[dylIT].append(bee.estimatedQualityTable[dylIT])

      # calculate new MEAN estimated quality for each field and store in the appropriate section of the history
      if bee.beeID == allActionValuesBee and meanUptakeCalcOverTimeOnOff == 1:
         for dylIU in range(1, (dylNumberOfFieldsInLandscape + 1)):
            bee.tempMeanEstCalc                =                float(sum(bee.dylEstimatedQualityHistory[dylIU]))                /
float(len(bee.dylEstimatedQualityHistory[dylIU]))
            bee.dylMeanEstimatedQualityHistory[dylIU].append(bee.tempMeanEstCalc)

      colony.append(bee)                                       # add the bee to the colony
      ##print "Bee " + str(bee.beeID) + " added to colony at GLOBAL CLOCK = " + str(globalClock)

      timeOfNextNewBee = globalClock + injectionInterval                  # the next new bee will be injected after the injection
interval has elapsed (as long as not at max number of bees)

      beeZeroEstimateAccuracy = open('GraphData/beeZeroEstimateAccuracy.txt','w')      # open file - will be written to throughout

      estQualsOverTime = open('GraphData/BEEZEROestQualsOverTime.txt','w')                  # open file - will be written to
throughout

      while beesAlive == 'true' and allFieldsEmpty == 'false' and cutOff == 'false':                  # whilst there are bees and FULL
flowers left in the simulation (and the simulation is not yet to be terminated)
         ##print "----------------------------------------"
         ##print "GLOBAL CLOCK = " + str(globalClock)
         #for ia in colony:
            ##print "BEE " + str(ia.beeID) + " @@ STATE = " + str(ia.state) + " @@ TO COMPLETE AT = " +
str(ia.completionTimeOfState) + " @@ in " + str(ia.destinationField)

         # empty the depletion tally
         dylDepletionTally = []

         # empty the chosen flower log
```

```
        chosenFlowerLog = []

        if len(colony) < maximumInjections:                    # if the colony is not already at its maximum
           if injectionInterval == 0 and globalClock == 0:        # if injection interval is 0, then all bees should be injected at start
              for iv in range(0, (maximumInjections - 1)):
                 bee = Bee()                                  # create the new bee

                 dylPassActionValueTable = []                 # reset passActionValueTable before use

                 # Initial action values are now populated as the distance values (in time units) specified between fields in the distance matrix.
                 for dylIO in range(0, (dylNumberOfFieldsInLandscape + 1)):
                    dylPassActionValueTable.append([])
                    for dylIP in range(0, (dylNumberOfFieldsInLandscape + 1)):
                       dylPassActionValueTable[dylIO].append(passDistanceMatrix[dylIO][dylIP])

                 dylPassEstimatedQualityTable = []            # reset passEstimatedQualityTable before use

                 # if colony is omniscient, then set initial estimated qualities as the actual field qualities
                 # if colony is normal, then set initial estimated qualities as the MEAN of the actual field qualities
                 if colonyOmniscience == 1:
                    resourceTable = open ('resourceTable.txt','r')

                    dylPassEstimatedQualityTable.append(0.0)       # nest DUMMY estimated quality
                    for dylIQ in range(1, (dylNumberOfFieldsInLandscape + 1)):
                       tempRead = resourceTable.readline()
                       tempRewIntFirst = int(resourceTable.readline())
                       tempRewInt = int(resourceTable.readline())
                       dylPassEstimatedQualityTable.append((float(tempRewIntFirst) / float(tempRewInt)))

                    resourceTable.close()
                 else:
                    tempRewRunningTally = 0.0

                    resourceTable = open('resourceTable.txt','r')

                    for dylIR in range(1, (dylNumberOfFieldsInLandscape + 1)):
                       tempRead = resourceTable.readline()
                       tempRewIntFirst = int(resourceTable.readline())
                       tempRewInt = int(resourceTable.readline())
                       tempRewFloatFirst = float(tempRewIntFirst) / float(tempRewInt)
                       tempRewRunningTally = float(tempRewRunningTally) + float(tempRewFloatFirst)

                    resourceTable.close()

                    tempRewFloat = float(tempRewRunningTally) / float(dylNumberOfFieldsInLandscape)

                    dylPassEstimatedQualityTable.append(0.0)       # nest DUMMY estimated quality
                    for dylIS in range(1, (dylNumberOfFieldsInLandscape + 1)):
                       dylPassEstimatedQualityTable.append(tempRewFloat)

                 # set up the bee with the initial parameters
                 bee.initialSetup(passLifespan, passSamplingInterval, passfixedNectarCapacity, passEpsilon, passAlpha, passBeta, passPlusRate, passMinusRate, dylPassActionValueTable, passDistanceMatrix, dylPassEstimatedQualityTable, passNestRateState)
                 bee.beeID = nextNewBeeID
                 bee.beeCreationTime = globalClock
                 nextNewBeeID = nextNewBeeID + 1                # next ID will be 1 higher
                 bee.remainingNectarCapacity = bee.fixedNectarCapacity
                 bee.state = "AT NEST"
                 bee.completionTimeOfState = globalClock + timeToCompleteATNEST

                 # reset of lists
                 bee.dylEstimatedQualityHistory = []
                 bee.dylMeanEstimatedQualityHistory = []

                 # INITIALISATION of lists
                 for dylIX in range(0, (dylNumberOfFieldsInLandscape + 1)):
                    bee.dylEstimatedQualityHistory.append([])
                    bee.dylMeanEstimatedQualityHistory.append([])

                 # append initial estimated qualities to history for each field
                 for dylIT in range(1, (dylNumberOfFieldsInLandscape + 1)):
                    bee.dylEstimatedQualityHistory[dylIT].append(bee.estimatedQualityTable[dylIT])

                 # calculate new MEAN estimated quality for each field and store in the appropriate section of the history
                 if bee.beeID == allActionValuesBee and meanUptakeCalcOverTimeOnOff == 1:
                    for dylIU in range(1, (dylNumberOfFieldsInLandscape + 1)):
```

```
                    bee.tempMeanEstCalc = float(sum(bee.dylEstimatedQualityHistory[dylIU])) /
float(len(bee.dylEstimatedQualityHistory[dylIU]))
                    bee.dylMeanEstimatedQualityHistory[dylIU].append(bee.tempMeanEstCalc)

                colony.append(bee)                          # add the bee to the colony
                ##print "Bee " + str(bee.beeID) + " added to colony at GLOBAL CLOCK = " + str(globalClock)

            if globalClock == timeOfNextNewBee and injectionInterval != 0:          # if it is time for a new bee
                bee = Bee()                                 # create the new bee

                dylPassActionValueTable = []                # reset passActionValueTable before use

                # Initial action values are now populated as the distance values (in time units) specified between fields in the distance
matrix.
                for dylIO in range(0, (dylNumberOfFieldsInLandscape + 1)):
                    dylPassActionValueTable.append([])
                    for dylIP in range(0, (dylNumberOfFieldsInLandscape + 1)):
                        dylPassActionValueTable[dylIO].append(passDistanceMatrix[dylIO][dylIP])

                dylPassEstimatedQualityTable = []           # reset passEstimatedQualityTable before use

                # if colony is omniscient, then set initial estimated qualities as the actual field qualities
                # if colony is normal, then set initial estimated qualities as the MEAN of the actual field qualities
                if colonyOmniscience == 1:
                    resourceTable = open ('resourceTable.txt','r')

                    dylPassEstimatedQualityTable.append(0.0)    # nest DUMMY estimated quality
                    for dylIQ in range(1, (dylNumberOfFieldsInLandscape + 1)):
                        tempRead = resourceTable.readline()
                        tempRewIntFirst = int(resourceTable.readline())
                        tempRewInt = int(resourceTable.readline())
                        dylPassEstimatedQualityTable.append((float(tempRewIntFirst) / float(tempRewInt)))

                    resourceTable.close()
                else:
                    tempRewRunningTally = 0.0

                    resourceTable = open('resourceTable.txt','r')

                    for dylIR in range(1, (dylNumberOfFieldsInLandscape + 1)):
                        tempRead = resourceTable.readline()
                        tempRewIntFirst = int(resourceTable.readline())
                        tempRewInt = int(resourceTable.readline())
                        tempRewFloatFirst = float(tempRewIntFirst) / float(tempRewInt)
                        tempRewRunningTally = float(tempRewRunningTally) + float(tempRewFloatFirst)

                    resourceTable.close()

                    tempRewFloat = float(tempRewRunningTally) / float(dylNumberOfFieldsInLandscape)

                    dylPassEstimatedQualityTable.append(0.0)    # nest DUMMY estimated quality
                    for dylIS in range(1, (dylNumberOfFieldsInLandscape + 1)):
                        dylPassEstimatedQualityTable.append(tempRewFloat)

                # set up the bee with the initial parameters
                bee.initialSetup(passLifespan, passSamplingInterval, passfixedNectarCapacity, passEpsilon, passAlpha, passBeta,
passPlusRate, passMinusRate, dylPassActionValueTable, passDistanceMatrix, dylPassEstimatedQualityTable, passNestRateState)
                bee.beeID = nextNewBeeID
                bee.beeCreationTime = globalClock
                nextNewBeeID = nextNewBeeID + 1                 # next ID will be 1 higher
                bee.remainingNectarCapacity = bee.fixedNectarCapacity
                bee.state = "AT NEST"
                bee.completionTimeOfState = globalClock + timeToCompleteATNEST

                # reset of lists
                bee.dylEstimatedQualityHistory = []
                bee.dylMeanEstimatedQualityHistory = []

                # INITIALISATION of lists
                for dylIX in range(0, (dylNumberOfFieldsInLandscape + 1)):
                    bee.dylEstimatedQualityHistory.append([])
                    bee.dylMeanEstimatedQualityHistory.append([])

                # append initial estimated qualities to history for each field
                for dylIT in range(1, (dylNumberOfFieldsInLandscape + 1)):
                    bee.dylEstimatedQualityHistory[dylIT].append(bee.estimatedQualityTable[dylIT])

                # calculate new MEAN estimated quality for each field and store in the appropriate section of the history
```

```
            if bee.beeID == allActionValuesBee and meanUptakeCalcOverTimeOnOff == 1:
               for dylIU in range(1, (dylNumberOfFieldsInLandscape + 1)):
                  bee.tempMeanEstCalc = float(sum(bee.dylEstimatedQualityHistory[dylIU])) /
float(len(bee.dylEstimatedQualityHistory[dylIU]))
                  bee.dylMeanEstimatedQualityHistory[dylIU].append(bee.tempMeanEstCalc)

            colony.append(bee)                        # add the bee to the colony
            ##print "Bee " + str(bee.beeID) + " added to colony at GLOBAL CLOCK = " + str(globalClock)

            timeOfNextNewBee = globalClock + injectionInterval      # set the next time of a new bee according to the injection
interval

         # loop through each bee in the colony - if they are due to complete a state, then the
         # relevant actions are performed
         for probedBee in colony:
            # if the bee has not had an initial calculation of action values yet, a calculation is made, and the flag is set to
            # indicate this has now been done
            if probedBee.initialActionValueCalculationMade == 0:
               probedBee.recalculateActionValues()
               probedBee.initialActionValueCalculationMade = 1

            ##print "Bee ID: " + str(probedBee.beeID) + " is in state " + str(probedBee.state)

            # add 1 to the relevant cell in the transition matrix
            tempTransMatValue = (transitionMatrix[probedBee.currentField])[probedBee.destinationField]
            tempTransMatValue = tempTransMatValue + 1
            (transitionMatrix[probedBee.currentField])[probedBee.destinationField] = tempTransMatValue

            if probedBee.completionTimeOfState == globalClock:
               if probedBee.state == "Traveling":          # if bee traveling and due to complete
                  ##print "Bee " + str(probedBee.beeID) + " actioned TRAVELING"
                  if probedBee.destinationField == 0:       # if bee returning to the nest
                     beeXTour = open('tourFiles/bee' + str(probedBee.beeID) + 'Tour.txt','a')
                     beeXTour.write("-9999999")
                     beeXTour.write("\n")
                     beeXTour.close()

                     probedBee.completedOneBout = 1
                     probedBee.boutCompleted = 'true'          # indicate that this represents the end of the bout

                     probedBee.state = "AT NEST"               # change state to 'at nest'
                     probedBee.currentField = 0
                     probedBee.destinationField = -1
                     probedBee.completionTimeOfState = globalClock + timeToCompleteATNEST    # work out completion time of
state

                     probedBee.timeFinishedBout = globalClock     # since the bout is finished, log the time of completion

                     ##print "!!!!!  Bout completed for BEE ID : " + str(probedBee.beeID)

                     probedBee.traffic.append(probedBee.trafficCount)        # append the traffic in the bout for the bee to the bee's
total life traffic list
                     colonyTraffic.append(probedBee.trafficCount)          # also append to the colony's total traffic list

                     # add the tally of field visits (for each field) in this bout to the life list
                     for dylIV in range(1, (dylNumberOfFieldsInLandscape + 1)):
                        probedBee.dylFieldVisitsPerBout[dylIV].append(probedBee.dylFieldBoutCount[dylIV])

                     # calculate bee's performance score for the bout
                     probedBee.boutPerformance = probedBee.nectarExtractedInBout / ((probedBee.timeFinishedBout -
probedBee.timeStartedBout) / probedBee.samplingInterval)
                     probedBee.performance.append(probedBee.boutPerformance)          # add the performance score to the log
                     colonyPerformance.append(probedBee.boutPerformance)              # also append to the colony's overall
performance log

                     probedBee.foragingBoutTally = probedBee.foragingBoutTally + 1    # increment the number of foraging bouts
completed
                     if probedBee.foragingBoutTally >= probedBee.lifespan:          # if the bee has been on the maximum number of
foraging bouts
                        probedBee.state = "Dead"                        # change the bee's state to 'Dead'
                        probedBee.completionTimeOfState = -1            # add dummy completion time so never actioned
                        ##print "Bee " + str(probedBee.beeID) + " has died"
                        numberOfDeadBees = numberOfDeadBees + 1         # increment the tally of deaths

                     # find the number of unique fields visited by the bee in this bout, then add it to the maintained list for that bee
                     #probedBee.listOfUniquesPerBout.append((probedBee.dylFieldVisitedInBout.count(1)))

                     # reset bout-based unique field counter for the bee
```

```
                    for ia,ib in enumerate(probedBee.dylFieldVisitedInBout):
                        probedBee.dylFieldVisitedInBout[ia] = 0

                    # if all of the bees are dead, then alert the fact that there are no more living bees in the simulation
                    if numberOfDeadBees == len(colony):
                        beesAlive = 'false'

                    probedBee.remainingNectarCapacity = probedBee.fixedNectarCapacity   # reset the remaining nectar capacity
                    probedBee.boutPerformance = 0                                       # reset the bout performance score
                    probedBee.decisionsInBout = 0                                       # reset the bout decisions tally
                    probedBee.trafficCount = 0                                          # reset the bout traffic tally

                    # reset the bee's action-values and estimated qualities to defaults with probaility omega
                    naivetyAssessment = random.uniform(0.0,1.0)

                    # if it has been determined that the bee should be switched to naivety, then reset action-values and estimated
qualities
                    if naivetyAssessment <= omega:
                        # reset action-values
                        for i in range(0, len(passDistanceMatrix)):
                            for j in range(0, len(passDistanceMatrix[i])):
                                probedBee.dylActionValueTable[i][j] = float(passDistanceMatrix[i][j])

                        # reset estimated qualities
                        for i in range(0, len(originalEstimatedQualityTable)):
                            probedBee.estimatedQualityTable[i] = float(originalEstimatedQualityTable[i])

                        # add this reset to the naivety report
                        naivetyReport = open('naivetyReport.txt','a')

                        naivetyReport.write(str(probedBee.beeID))
                        naivetyReport.write(",")
                        naivetyReport.write(str(globalClock))
                        naivetyReport.write("\n")

                        naivetyReport.close()

                    # reset the counters
                    for dylIW in range(1, (dylNumberOfFieldsInLandscape + 1)):
                        probedBee.dylFieldBoutCount[dylIW] = 0

                    probedBee.full = 'false'                                    # indicate that the bee is no longer full
                    probedBee.nectarExtractedInBout = 0                         # reset the bout nectar collected record
                else:                                                   # if bee not traveling to nest
                    beeXTour = open('tourFiles/bee' + str(probedBee.beeID) + 'Tour.txt','a')
                    beeXTour.write("-9999999")
                    beeXTour.write("\n")
                    beeXTour.close()

                    probedBee.state = "Foraging"                    # change the bee's state to 'foraging'
                    probedBee.completionTimeOfState = globalClock + probedBee.samplingInterval      # set the completion time
according to the sampling interval

            if probedBee.completionTimeOfState == globalClock:
                if probedBee.state == "AT NEST":                    # if bee at nest and due to complete
                    ##print "Bee " + str(probedBee.beeID) + " actioned AT NEST"

                    # if colony omniscience is on, then give the bee an update of the true qualities and recalculate action values before a
decision is made
                    if colonyOmniscience == 1:
                        resourceTable = open ('resourceTable.txt','r')

                        for dylIX in range(1, (dylNumberOfFieldsInLandscape + 1)):
                            tempRead = resourceTable.readline()
                            dylFull[dylIX] = int(resourceTable.readline())
                            dylTotal[dylIX] = int(resourceTable.readline())

                        resourceTable.close()

                        # set estimates as actual qualities
                        for dylIY in range(1, (dylNumberOfFieldsInLandscape + 1)):
                            probedBee.estimatedQualityTable[dylIY] = float(dylFull[dylIY]) / float(dylTotal[dylIY])

                        # if an estimated quality of a field is 0, then the estimated quality is replaced with a VERY small number to
                        # avoid action value calculations that attempt to divide by 0
                        for dylIZ in range(1, (dylNumberOfFieldsInLandscape + 1)):
                            if probedBee.estimatedQualityTable[dylIZ] == 0:
                                probedBee.estimatedQualityTable[dylIZ] = 0.00000000000000000001
```

```
                probedBee.recalculateActionValues()

            probedBee.choosePolicyType()                # choose greedy / not-necc. greedy
            probedBee.selectActionValue()               # select an action value
            probedBee.state = "Traveling"               # change state to 'traveling'

            # find the distance (in time units) to the selected destination, to work out when the 'traveling'
            # state should complete
            probedBee.completionTimeOfState = globalClock + (probedBee.distanceMatrix[probedBee.currentField][probedBee.destinationField])

            probedBee.timeStartedBout = globalClock     # log the time a foraging bout was started

            beeXTour = open ('tourFiles/bee' + str(probedBee.beeID) + 'Tour.txt','a')
            beeXTour.write("-")
            beeXTour.write(str(probedBee.destinationField))
            beeXTour.write("\n")
            beeXTour.close()

        if probedBee.completionTimeOfState == globalClock:
            if probedBee.state == "Foraging":           # if bee foraging and due to complete
            # indicates that the destination field is now being visited by the bee (and therefore has been visited at least once by the bee)
            probedBee.dylFieldVisitedInLife[probedBee.destinationField] = 1
            probedBee.dylFieldVisitedInBout[probedBee.destinationField] = 1
            ##print "Bee " + str(probedBee.beeID) + " actioned FORAGING"

            # increment the bee's personal running residence counter by 1 flower visit
            probedBee.runningResidenceCounter = probedBee.runningResidenceCounter + 1

            # apply a time stamp in the appropriate location in allBeePatchTimeStamps
            allBeePatchTimeStamps[probedBee.beeID][(probedBee.destinationField - 1)] = globalClock

            # while an acceptable flower has not been chosen
            while acceptableFlowerChosen == 0:
                probedBee.selectFlower(explicitFlowerAllFields)         # select a flower to sample

                if probedBee.flowerTypeToSample == "FULL":
                    ##print "Bee " + str(probedBee.beeID) + " sampled a FULL flower."
                    # If depletion and replenishment is enabled, incrememnt the tally of flowers depleted and mark for depletion
                    if useDepletionAndReplenishment == 1:
                        flowersDepletedFromSystem = flowersDepletedFromSystem + 1
                        dylDepletionTally.append([(probedBee.explicitFlowerToSample[0]), (probedBee.explicitFlowerToSample[1])])
                    chosenFlowerLog.append([(probedBee.explicitFlowerToSample[0]), (probedBee.explicitFlowerToSample[1])])
                else:
                    chosenFlowerLog.append([(probedBee.explicitFlowerToSample[0]), (probedBee.explicitFlowerToSample[1])])
                    if probedBee.beeID == allActionValuesBee:
                        timesWhenBeeSampledEMPTYFlowers.append(globalClock)

                # go through and count the number of matches in the chosen flower log with the probed bee's chosen field, and the probed bee's chosen flower
                for iz in range(0, len(chosenFlowerLog)):
                    if chosenFlowerLog[iz][0] == probedBee.destinationField:
                        fieldMatches = fieldMatches + 1
                        if chosenFlowerLog[iz][1] == probedBee.explicitFlowerToSample[1]:
                            flowerMatches = flowerMatches + 1

                #print "Field Matches : " + str(fieldMatches)
                #print chosenFlowerLog

                # if all of the flowers in the field are occupied, then there will be (no. of flowers + 1) matching fields in the depletion tally
                # ("+ 1" as this latest attempt will have been added, causing an overflow)
                # in this event, set the bee's estimate of the field to 0, and if the flower the bee had chosen is FULL, tell the bee it was empty
                # so that the bee will not extract any nectar
                if fieldMatches == len(explicitFlowerAllFields[probedBee.destinationField]) + 1:
                    #print "All occupied"
                    probedBee.estimatedQualityTable[probedBee.destinationField] = 0.00000000000000000001
                    if probedBee.flowerTypeToSample == "FULL":
                        probedBee.flowerTypeToSample = "EMPTY"
                        # update the tallies appropriately
                        probedBee.fullFlowerSelections = probedBee.fullFlowerSelections - 1
                        probedBee.emptyFlowerSelections = probedBee.emptyFlowerSelections + 1
                        flowersDepletedFromSystem = flowersDepletedFromSystem - 1
                        if useDepletionAndReplenishment == 1:
```

```
                        # if the flower was full, then an entry would have been added to the end of the depletion tally
                        # remove it now
                        dylDepletionTally.pop()
                    # remove the latest entry from the chosen flower log
                    chosenFlowerLog.pop()
                    # indicate an acceptable flower chosen (in this case, a virtual EMPTY flower as there are no choosable flowers
                    # in the patch, so there is no point in selecting again at this time)
                    acceptableFlowerChosen = 1
                else:
                    if flowerMatches > 1:
                        #print "Clash in selection"
                        # if there is a clash in flower selection, and there are other flowers left to choose (unoccupied)
                        if probedBee.flowerTypeToSample == "FULL":
                            probedBee.fullFlowerSelections = probedBee.fullFlowerSelections - 1        # update tally
                            if useDepletionAndReplenishment == 1:
                                # if the flower was full, then an entry would have been added to the end of the depletion tally
                                # remove it now
                                dylDepletionTally.pop()
                        # remove the latest entry from the chosen flower log
                        chosenFlowerLog.pop()
                        if probedBee.flowerTypeToSample == "EMPTY":
                            probedBee.fullFlowerSelections = probedBee.emptyFlowerSelections - 1        # update tally
                    else:
                        #print "No clashes"
                        # if there are no flower clashes, and all flowers not occupied, then the flower chosen was acceptable
                        acceptableFlowerChosen = 1

                # reset the match tallies
                fieldMatches = 0
                flowerMatches = 0

            acceptableFlowerChosen = 0                    # reset the flag

            probedBee.extractNectar()                    # provide the bee with the relevant amount of nectar

            # if colony omniscience is on, then give the bee an update of the true qualities and recalculate action values before a
decision is made
            if colonyOmniscience == 1:
                resourceTable = open ('resourceTable.txt','r')

                for dylIX in range(1, (dylNumberOfFieldsInLandscape + 1)):
                    tempRead = resourceTable.readline()
                    dylFull[dylIX] = int(resourceTable.readline())
                    dylTotal[dylIX] = int(resourceTable.readline())

                resourceTable.close()

                # set estimates as actual qualities
                for dylIY in range(1, (dylNumberOfFieldsInLandscape + 1)):
                    probedBee.estimatedQualityTable[dylIY] = float(dylFull[dylIY]) / float(dylTotal[dylIY])

                # if an estimated quality of a field is 0, then the estimated quality is replaced with a VERY small number to
                # avoid action value calculations that attempt to divide by 0
                for dylIZ in range(1, (dylNumberOfFieldsInLandscape + 1)):
                    if probedBee.estimatedQualityTable[dylIZ] == 0:
                        probedBee.estimatedQualityTable[dylIZ] = 0.00000000000000000001

                probedBee.recalculateActionValues()
            if colonyOmniscience != 1:
                probedBee.recalculateEstimatedQuality()            # recalculate estimated quality of the field in which the bee
foraged

            probedBee.recalculateActionValues()            # recalculate all action values

            probedBee.currentField = probedBee.destinationField        # indicate that the previously foraged field is now the
starting point for any potential moves
            probedBee.destinationField = -1                    # indicate that no decision has yet been made regarding the next
move

            # if the bee is full to capacity with nectar, then change the bee's state to 'Traveling' with a
            # destination of the nest
            if probedBee.full == 'true':
                probedBee.destinationField = 0
                probedBee.state = "Traveling"
                #probedBee.trafficCount = probedBee.trafficCount + 1        #NOT COUNTING RETURN TRIP AT THE
MOMENT

                # now that the bee has left, record the residence in this patch
```

```
                    # find the patch type of the field in question
                    tempInt = 0

                    for ij in range(0, len(patchTypeNominations)):
                        for ik in range(0, len(patchTypeNominations[ij])):
                            if patchTypeNominations[ij][ik] == probedBee.currentField:                # if field matches
                                tempInt = ij                                # note the outer index as this corresponds with patch type

                    # add the visit's residence to the bee's personal log
                    probedBee.residencePerPatch.append([probedBee.runningResidenceCounter,                    probedBee.currentField,
typesOfPatch[tempInt]])

                    tempInt = 0

                    probedBee.runningResidenceCounter = 0                    # reset the bee's running residence counter

                    probedBee.completionTimeOfState        =        globalClock        +        probedBee.samplingInterval        +
(probedBee.distanceMatrix[probedBee.currentField][probedBee.destinationField])

                    beeXTour = open ('tourFiles/bee' + str(probedBee.beeID) + 'Tour.txt','a')
                    beeXTour.write("-8888888")
                    beeXTour.write("\n")
                    beeXTour.close()
                else:                                # if the bee is not full to capacity
                    probedBee.choosePolicyType()                    # choose between greedy or not-necc. greedy
                    probedBee.selectActionValue()                    # select an action value

                    # if the bee is staying at its current field, then the bee re-enters a state of foraging
                    # but if a move is to be made, then the bee enters a state of 'traveling'
                    if probedBee.currentField == probedBee.destinationField:
                        probedBee.state = "Foraging"
                        probedBee.completionTimeOfState = globalClock + probedBee.samplingInterval
                    else:
                        probedBee.state = "Traveling"

                        # now that the bee has left, record the residence in this patch
                        # find the patch type of the field in question
                        tempInt = 0

                        for ij in range(0, len(patchTypeNominations)):
                            for ik in range(0, len(patchTypeNominations[ij])):
                                if patchTypeNominations[ij][ik] == probedBee.currentField:                # if field matches
                                    tempInt = ij                                # note the outer index as this corresponds with patch type

                        # add the visit's residence to the bee's personal log
                        probedBee.residencePerPatch.append([probedBee.runningResidenceCounter,                    probedBee.currentField,
typesOfPatch[tempInt]])

                        tempInt = 0

                        probedBee.runningResidenceCounter = 0                    # reset the bee's running residence counter

                        probedBee.completionTimeOfState        =        globalClock        +        probedBee.samplingInterval        +
(probedBee.distanceMatrix[probedBee.currentField][probedBee.destinationField])
                        beeXTour = open ('tourFiles/bee' + str(probedBee.beeID) + 'Tour.txt','a')
                        beeXTour.write("-")
                        beeXTour.write(str(probedBee.destinationField))
                        beeXTour.write("\n")
                        beeXTour.close()

            # for each of this bee's patch quality estimates, update according to the appropriate estimated replenishment mechanism
            # if mechanism is EXPONENTIAL MODEL
            if estimatedReplenishmentModel == 0:
                # for each patch estimate
                for ia in range(1, len(probedBee.estimatedQualityTable)):
                    # work out the time difference between the current time and the time of last visit to the patch
                    # (note - (ia - 1) as allBeePatchTimeStamps has patch 1 = index 0)
                    tempTimeDifference = int(globalClock) - int(allBeePatchTimeStamps[probedBee.beeID][(ia - 1)])

                    # recalculate the estimated quality for the patch, factoring in estimated replenishment using an exponential
mechanism
                    probedBee.estimatedQualityTable[ia]        =        float(probedBee.estimatedQualityTable[ia])        *        float((math.pow(gamma,
tempTimeDifference)))

            # if mechanism is LINEAR MODEL
            if estimatedReplenishmentModel == 1:
                # for each patch estimate
                for ib in range(1, len(probedBee.estimatedQualityTable)):
```

```
                probedBee.estimatedQualityTable[ib] = float(probedBee.estimatedQualityTable[ib]) + float(gamma)

            # if option set to cap estimated qualities to 1.0, go through and cap as appropriate
            if capEstQual == 1:
                for ic in range(1, len(probedBee.estimatedQualityTable)):
                    if probedBee.estimatedQualityTable[ic] > 1.0:
                        probedBee.estimatedQualityTable[ic] = 1.0

        ##print "SWEEP END - Depletion Constant = " + str(depletionConstant)
        # go through the colony and find out where each bee is to determine the bee density per patch at this time
        # reset the tempDensityEntry counts to 0 first
        for iz in range(0, len(tempDensityEntry)):
            tempDensityEntry[iz] = 0

        # for each bee, if it is currently foraging, then increment the relevant counter (for the corresponding patch) in
tempDensityEntry
        # (field - 1) used as tempDensityEntry does not record nest (which is field 0)
        for ia in colony:
            if ia.state == "Foraging":
                tempDensityEntry[(ia.destinationField - 1)] = tempDensityEntry[(ia.destinationField - 1)] + 1

        # append the list for this time unit to the main list over time
        densityPerPatch.append(tempDensityEntry)

        # reset the temp list
        tempDensityEntry = []

        # set up tempDensityEntry with dummy entries according to the number of patches in the landscape
        for ia in range(0, dylNumberOfFieldsInLandscape):
            tempDensityEntry.append(0)

        # write to the estQualsOverTime file bee 0's current estimated qualities for each patch
        for iq in colony:
            if iq.beeID == 0:
                for iz in range(1, (dylNumberOfFieldsInLandscape + 1)):
                    estQualsOverTime.write(str(iq.estimatedQualityTable[iz]))
                    estQualsOverTime.write(",")

        estQualsOverTime.write("\n")

        # read in current FULL / Total flower quantities from all entries in resource table
        # subtract FULL flowers from each according to the relevant tally
        resourceTable = open ('resourceTable.txt','r')

        for dylIX in range(1, (dylNumberOfFieldsInLandscape + 1)):
            tempRead = resourceTable.readline()
            dylFull[dylIX] = int(resourceTable.readline())
            dylTotal[dylIX] = int(resourceTable.readline())

        resourceTable.close()

        if useDepletionAndReplenishment == 1:
            # look for duplicates in the depletion tally (which could arise if more than 1 bee samples the same flower)
            # and delete all but one of the entries (leaving no duplicates)
            for ip,ipp in enumerate(dylDepletionTally):
                for iq, iqq in enumerate(dylDepletionTally):
                    if dylDepletionTally[ip] == dylDepletionTally[iq] and ip != iq:
                        del dylDepletionTally[iq]

            ##print "Fulls BEFORE : " + str(dylFull)

            # go through each entry in the depletion tally and set the specified field's specified flower's reward to 0
            # update total number of full flowers in the fields affected
            # also, add the depleted explicit flowers to the replenishment log, marking them for future replenishment
            for dylIA in range(0, len(dylDepletionTally)):
                explicitFlowerAllFields[(dylDepletionTally[dylIA][0])][(dylDepletionTally[dylIA][1])] = 0
                dylFull[(dylDepletionTally[dylIA][0])] = dylFull[(dylDepletionTally[dylIA][0])] - 1

                # find the patch type of the field in question
                for ij in range(0, len(patchTypeNominations)):
                    for ik in range(0, len(patchTypeNominations[ij])):
                        if patchTypeNominations[ij][ik] == dylDepletionTally[dylIA][0]:                # if field matches
                            tempInt = ij                                # note the outer index as this corresponds with patch type

                for probedBee in colony:
                    tempSamplingInterval = int(probedBee.samplingInterval)
```

```
                replenishmentLog.append([(dylDepletionTally[dylIA][0]), (dylDepletionTally[dylIA][1]), int(globalClock + tempSamplingInterval + dylReplenishmentInterval[tempInt])])

                tempInt = 0

            ##print "Fulls AFTER : " + str(dylFull)

        # if depletion sets number of FULL flowers to less than 0, then override to be 0
        # IT SHOULD NO LONGER DO THIS WITH EXPLICIT FLOWER MODELLING, BUT HERE JUST IN CASE
        for dylIB in range(1, (dylNumberOfFieldsInLandscape + 1)):
            if dylFull[dylIB] < 0:
                dylFull[dylIB] = 0

        # stores FULL flowers before replenishment
        for dylIC in range(1, (dylNumberOfFieldsInLandscape + 1)):
            tempOldFull = tempOldFull + dylFull[dylIC]

        ##print "REP - Total Standing Crop BEFORE : " + str(tempOldFull)

        if useDepletionAndReplenishment == 1:
            # go through each entry in the replenishment log - if the entry indicates that a replenishment is due
            # then the corresponding explicit flower is refilled in the explicitFlowerAllFields array
            # update total number of full flowers in the fields affected
            # finally, mark the replenishment log entry for deletion by setting the time of replenishment to -1
            for ie in range(0, len(replenishmentLog)):
                if replenishmentLog[ie][2] == globalClock:
                    explicitFlowerAllFields[(replenishmentLog[ie][0])][(replenishmentLog[ie][1])] = 1
                    dylFull[(replenishmentLog[ie][0])] = dylFull[(replenishmentLog[ie][0])] + 1
                    replenishmentLog[ie][2] = -1

            # go through each entry in the replenishment log, and delete those entries marked for deletion
            for ig, ih in enumerate(replenishmentLog):
                if replenishmentLog[ig][2] == -1:
                    del replenishmentLog[ig]

        ##print "REP - FULLs AFTER : " + str(dylFull)

        # if the sudden resource injection facility is turned on
        if suddenResourceInjectionOnOff == 1:
            # and it's time to inject
            if suddenResourceInjectionTime == globalClock:
                # go through the array of explicit flowers for the specified field, and keep filling emptys until no more
                # injections specified, or no more flowers left to fill
                for ik in range(0, len(explicitFlowerAllFields[suddenResourceInjectionField])):
                    if explicitFlowerAllFields[suddenResourceInjectionField][ik] == 0 and suddenResourceInjectionAmount > 0:
                        explicitFlowerAllFields[suddenResourceInjectionField][ik] = 1
                        suddenResourceInjectionAmount = suddenResourceInjectionAmount - 1

                # print message if could not make all of the specified number of flower injections
                if suddenResourceInjectionAmount > 0:
                    print "Resource Injection was too large.  Injections made where possible."

                # now add to the appropriate field's full flower tally
                dylFull[suddenResourceInjectionField] = dylFull[suddenResourceInjectionField] + suddenResourceInjectionAmount

        # if replenishment has caused number of full flowers to exceed total, then cap back to total
        # IT SHOULD NO LONGER DO THIS WITH EXPLICIT FLOWER MODELLING, BUT HERE JUST IN CASE
        for dylIE in range(1, (dylNumberOfFieldsInLandscape + 1)):
            if dylFull[dylIE] > dylTotal[dylIE]:
                dylFull[dylIE] = dylTotal[dylIE]

        # stores FULL flowers after replenishment
        for dylIF in range(1, (dylNumberOfFieldsInLandscape + 1)):
            tempNewFull = tempNewFull + dylFull[dylIF]

        newFlowersAddedToSystem = newFlowersAddedToSystem + (tempNewFull - tempOldFull)          # calculate number of new flowers added to system
        ##print "NEW FULL = " + str(tempNewFull)
        ##print "OLD FULL = " + str(tempOldFull)
        ##print "NFATS ---- " + str(newFlowersAddedToSystem)

        # if all fields are now depleted of FULL flowers, then boolean is set to true
        if dylFull.count(0) == len(dylFull):
            allFieldsEmpty = 'true'

        # calculate and store the total landscape standing crop at this time
        landscapeStandingCropOverTime.append(sum(dylFull))
```

```
        # calculate and store the average landscape standing crop (across the fields) at this time
        landscapeStandingCropOverTimeAveAcrossFields.append((float(sum(dylFull)) / 3.0))

        # rewrite the resource table with updated resource value
        resourceTable = open ('resourceTable.txt','w')

        for dylIB in range(1, (dylNumberOfFieldsInLandscape + 1)):
           resourceTable.write(("Field " + str(dylIB)))
           resourceTable.write("\n")
           resourceTable.write(str(dylFull[dylIB]))
           resourceTable.write("\n")
           resourceTable.write(str(dylTotal[dylIB]))
           resourceTable.write("\n")

        resourceTable.close()

        # calculate standing crop (field quality) range at this time, based on max and min qualities
        # *****IMPORTANT - these calcs are based on single field capacity across landscape
        dylTempMaxQuality = float(max(dylFull)) / float(dylTotal[1])
        dylTempMinQuality = float(min(dylFull)) / float(dylTotal[1])

        dylRangeInStandingCropOverTime.append((dylTempMaxQuality - dylTempMinQuality))      # append range for this time
unit to list

        # append number of full flowers at this time in each field to the appropriate section of the history
        for dylIG in range(1, (dylNumberOfFieldsInLandscape + 1)):
           dylFullOverTime[dylIG].append(dylFull[dylIG])

        if allFieldsEmpty == 'true':
           allFieldsEmpty = 'true'
           print "***** Total Landscape Nectar Depletion - No more full flowers"
           print "SIMULATION TERMINATING"

        # add estimated qualities of each field for each bee in the colony to the history
        for ig in colony:
           for dylIH in range(1, (dylNumberOfFieldsInLandscape + 1)):
              ig.dylEstimatedQualityHistory[dylIH].append(ig.estimatedQualityTable[dylIH])

           # calculate and store new mean estimated quality (over time so far) for each field
           if ig.beeID == allActionValuesBee and meanUptakeCalcOverTimeOnOff == 1:
              for dylII in range(1, (dylNumberOfFieldsInLandscape + 1)):
                 ig.tempMeanEstCalc                =                float(sum(ig.dylEstimatedQualityHistory[dylII]))                /
float(len(ig.dylEstimatedQualityHistory[dylII]))
                 ig.dylMeanEstimatedQualityHistory[dylII].append(ig.tempMeanEstCalc)

        # append the current location of each bee to the tours
        for ih in colony:
           if ih.boutCompleted == 'false':
              ih.tour.append(ih.currentField)                # add the current field to the tour
              # append current location to the tour file
              beeXTour = open('tourFiles/bee' + str(ih.beeID) + 'Tour.txt', 'a')
              beeXTour.write(str(ih.currentField))
              beeXTour.write("\n")
              beeXTour.close()
           else:
              if timeToCompleteATNEST == 0:
                 ih.allTours.append(ih.tour)                # add the bout tour to the list of tours for the bee
                 ih.allToursMerged.extend(ih.tour)              # add the bout tour to the single list of all locations
                 ih.tour = []                          # reset the tour
              else:
                 ih.tour.append(ih.currentField)                # add the current field (nest) to the tour as the final location of the bout
                 # append current location to the bee tour file
                 beeXTour = open('tourFiles/bee' + str(ih.beeID) + 'Tour.txt', 'a')
                 beeXTour.write(str(ih.currentField))
                 beeXTour.write("\n")
                 beeXTour.close()
                 ih.allTours.append(ih.tour)                # add the bout tour to the list of tours for the bee
                 ih.allToursMerged.extend(ih.tour)              # add the bout tour to the single list of all locations
                 ih.tour = []                          # reset the tour
              ih.boutCompleted = 'false'

        # log the remaining capacity over time of the tracked bee
        for ii in colony:
           if ii.beeID == allActionValuesBee:
              ii.remainingCapacityHistory.append(ii.remainingNectarCapacity)

        globalClock = globalClock + 1              # increment the global clock
```

```
# if it has been defined by the user that the simulation should end now then do so
if simCutOffTime == globalClock:
   ##print "***** Simulation cut off at Global Clock = " + str(simCutOffTime)
   ##print "SIMULATION TERMINATED"
   cutOff = 'true'

tempBoutsTotalColony = 0                    # reset parameter before using

# add up the total number of bouts so far completed across the colony
for ia in colony:
   tempBoutsTotalColony = tempBoutsTotalColony + ia.foragingBoutTally

# work out the mean number of bouts completed across the colony at this point
tempMeanBouts = float(tempBoutsTotalColony) / float(len(colony))

# append this mean to the list
meanBoutsCompletedThroughClock.append(tempMeanBouts)

# write the current patch qualities to the patchQualitiesOverTime file
patchQualitiesOverTime = open('GraphData/patchQualitiesOverTime.txt','a')

# for each patch, calculate the quality from the explicit flower array and write the result to file, separated by commas
# (range is from 1 as we exclude the nest)
for iu in range(1, len(explicitFlowerAllFields)):
   tempFullReportToFile = explicitFlowerAllFields[iu].count(1)
   tempTotalReportToFile = len(explicitFlowerAllFields[iu])
   tempQualityReportToFile = float(tempFullReportToFile) / float(tempTotalReportToFile)
   patchQualitiesOverTime.write((str(tempQualityReportToFile) + ","))

   # find bee 0 and set accuracy indicator for the currently referenced patch accordingly
   for findZero in colony:
      if findZero.beeID == 0:
         if float(findZero.estimatedQualityTable[iu]) == (float(dylFull[iu]) / float(dylTotal[iu])):
            accuracyIndicator = 0
         if float(findZero.estimatedQualityTable[iu]) >= ((float(dylFull[iu]) / float(dylTotal[iu])) - 0.01) and
float(findZero.estimatedQualityTable[iu]) <= ((float(dylFull[iu]) / float(dylTotal[iu])) + 0.01):
            accuracyIndicator = 0
         else:
            if float(findZero.estimatedQualityTable[iu]) < (float(dylFull[iu]) / float(dylTotal[iu])):
               accuracyIndicator = -1
            if float(findZero.estimatedQualityTable[iu]) > (float(dylFull[iu]) / float(dylTotal[iu])):
               accuracyIndicator = 1
                     #print "PATCH    = " + str(iu)
                     #print "Estimate = " + str(findZero.estimatedQualityTable[iu])
                     #print "Actual   = " + str((float(dylFull[iu]) / float(dylTotal[iu])))
                     #print "Acc. Ind.= " + str(accuracyIndicator)
                     #print "----------------------"

   # write the accuracy indicator for the patch to the file
   beeZeroEstimateAccuracy.write(str(accuracyIndicator))
   beeZeroEstimateAccuracy.write(",")

# write a new line once all accuracy indicators for this time step written
beeZeroEstimateAccuracy.write("\n")

# write a new line once all field qualities calculated and written
patchQualitiesOverTime.write("\n")

# close the file
patchQualitiesOverTime.close()

# now that the global clock has ticked, calculate the rate of nectar uptake for all ALIVE bees in the colony
# and append to the bee's list of rate of uptake values
for ik in colony:
   if ik.state != "Dead":
      tempTimeExisted = globalClock - ik.beeCreationTime
      tempRateOfUptake = ik.totalNectarCollectedOverLife / tempTimeExisted
      ik.rateOfUptakeOverTime.append(tempRateOfUptake)

# work out the mean number of unique fields across the colony at this time, and append it to the list
#for ia in colony:
   #tempUniquesSum = tempUniquesSum + ia.dylFieldVisitedInLife.count(1)

#tempMeanUniques = float(tempUniquesSum) / float(len(colony))

#listOfMeanUniques.append(tempMeanUniques)

# reset the parameters
```

```
    #tempUniquesSum = 0.0
    #tempMeanUniques = 0.0

    # work out the mean performance across the colony at this time, and append it to the list
    #for ib in colony:
      #tempPerformancesSum = tempPerformancesSum + ib.rateOfUptakeOverTime[(len(ib.rateOfUptakeOverTime) - 1)]

    #tempMeanPerformances = float(tempPerformancesSum) / float(len(colony))

    #listOfMeanPerformances.append(tempMeanPerformances)

    # reset the parameters
    #tempPerformancesSum = 0.0
    #tempMeanPerformances = 0.0

    # reset parameters
    tempOldFull = 0
    tempNewFull = 0

    print globalClock

estQualsOverTime.close()

beeZeroEstimateAccuracy.close()

# write the densityPerPatch list to file
patchDensityOverTime = open('GraphData/patchDensityOverTime.txt','w')

for ia in range(0, len(densityPerPatch)):
    for ib in range(0, len(densityPerPatch[ia])):
        patchDensityOverTime.write(str(densityPerPatch[ia][ib]))
        patchDensityOverTime.write(",")

    patchDensityOverTime.write("\n")

patchDensityOverTime.close()

if dylMinimalLog == 0:
    #for ix in colony:
        ##print "Bee ID                : " + str(ix.beeID)
        ##print "GREEDY ACTIONS        : " + str(ix.greedyLifeCount)
        ##print "NOT-NECC GREEDY ACTIONS : " + str(ix.nnGreedyLifeCount)
        ##print "--------------------------"
        ##print "Total Actions =          " + str((ix.greedyLifeCount + ix.nnGreedyLifeCount))
        ##print ""

    meanColonyTraffic = float(sum(colonyTraffic)) / float(len(colonyTraffic))
    #print "...MEAN COLONY TRAFFIC PER BOUT     = " + str(round(meanColonyTraffic, 3))

    meanColonyPerformance = float(sum(colonyPerformance)) / float(len(colonyPerformance))
    #print "...MEAN COLONY PERFORMANCE PER BOUT = " + str(round(meanColonyPerformance, 3))

    colonyTotalTraffic = float(sum(colonyTraffic))
    #print "...TOTAL TRAFFIC IN SIMULATION      = " + str(colonyTotalTraffic)
    #print ""

    # add up and record total forages by the colony in each field
    for iz in colony:
        for dylIJ in range(1, (dylNumberOfFieldsInLandscape + 1)):
            dylTotalForages[dylIJ] = dylTotalForages[dylIJ] + iz.dylFieldForages[dylIJ]

    #for dylIK in range(1, (dylNumberOfFieldsInLandscape + 1)):
        #print "Total Forages in Field " + str(dylIK) + " : " + str(dylTotalForages[dylIK])

    #print "----------------------------"
    #print "TOTAL FORAGES IN LANDSCAPE = " + str(sum(dylTotalForages))

    # plot graphs

    # Mean Foraging Bouts Completed Over Time
    figure(1)
    plot((range(1,(globalClock + 1))),meanBoutsCompletedThroughClock)
    axis([1,(globalClock + 1),0,passLifespan])
    xlabel('Time (GLOBAL CLOCK)')
    ylabel('Mean Bouts Completed Across Colony')
    title('BOUTS COMPLETED OVER TIME')

    # add each bee's 'performance over time' log to the general list
```

```
        for ib in colony:
            listOfBeePerformanceRecords.append(ib.performance)

        #for dylIZ in colony:
            #if dylIZ.completedOneBout == 0:
                #print "Bee " + str(dylIZ.beeID) + " has not completed a bout."

        # find the maximum performance score achieved in the colony -
        # for each 'performance over time' list, check to see if its max is greater than the currently recorded colony max -
        # if so, update the colony max accordingly
        for ic in listOfBeePerformanceRecords:
            if max(ic) > maxPerformanceInColony:
                maxPerformanceInColony = max(ic)

        # identify field with highest number of FULL flowers at start of simulation
        # IMPORTANT - this will need to change to highest QUALITY if and when field Total Flowers can vary
        for dylIL in range(1, (dylNumberOfFieldsInLandscape + 1)):
            if dylStartingFull[dylIL] == max(dylStartingFull):
                highestStartingField = dylIL

        # identify the distance from the nest to the highest starting quality field
        distanceToHighestStartingField = colony[0].distanceMatrix[0][highestStartingField]

        # calculate the highest achievable performance based on an exclusive move to the best starting field, a stay there until full to
capacity, and then
        # a return to the nest
        highestAchievablePerformance = float(passfixedNectarCapacity) / (((2.0 * float(distanceToHighestStartingField)) +
float(passfixedNectarCapacity)) / passSamplingInterval)
        #print "Highest Achievable Performance = " + str(round(highestAchievablePerformance, 3))

        # performance over time
        figure(14)
        axhline(highestAchievablePerformance)
        for ie in listOfBeePerformanceRecords:
            plot(range(1,(len(ie) + 1)), ie)
        axis([1,passLifespan,0,1])
        xlabel('Foraging Bouts Completed')
        ylabel('Performance Score')
        title('Performance Over Time Per Bee')

        # build a list of the length of each bee performance record
        for ig in listOfBeePerformanceRecords:
            lengthOfBeePerformanceRecords.append(len(ig))

        # find the smallest number of bouts that a bee completed
        smallestCompletedBouts = min(lengthOfBeePerformanceRecords)

        # build list of mean performance scores for colony when all bees completed 1 bout, 2 bouts etc...
        for ih in range(0,smallestCompletedBouts):
            for ij in listOfBeePerformanceRecords:
                tempTotalPerformanceColony = tempTotalPerformanceColony + ij[ih]
            tempMeanCalc = tempTotalPerformanceColony / len(listOfBeePerformanceRecords)
            meanColonyPerformanceOverTime.append(tempMeanCalc)
            tempTotalPerformanceColony = 0.0

        # average performance of colony over time (mean of plots in figure(14))
        figure(15)
        axhline(highestAchievablePerformance)
        plot(range(1,(len(meanColonyPerformanceOverTime) + 1)), meanColonyPerformanceOverTime)
        axis([1,len(meanColonyPerformanceOverTime),0,1])
        xlabel('Foraging Bouts Completed By All Bees')
        ylabel('Mean Colony Performance Score')
        title('Mean Colony Performance Over Time')

        # rate of nectar uptake over time (for every tick of the global clock) per bee
        figure(16)
        for im in colony:
            plot(range(1,(len(im.rateOfUptakeOverTime) + 1)), im.rateOfUptakeOverTime)
        axis([1,globalClock,0,1])
        xlabel('Time Bee Has Existed')
        ylabel('Rate of Nectar Uptake')
        title('Rate of Nectar Uptake Per Bee Over Time')

        # calculate overall mean nectar uptake rate across colony
        meanOverallUptakeRate = 0
        tempLen = 0

        for iq in colony:
```

```
            tempSum = float(tempSum) + float(sum(iq.rateOfUptakeOverTime))
            tempLen = float(tempLen) + float(len(iq.rateOfUptakeOverTime))

        meanOverallUptakeRate = float(tempSum) / float(tempLen)

        #print "###MEAN Nectar Uptake Rate Across Colony = " + str(round(meanOverallUptakeRate, 3))

        # build a list of ALL rate of uptake values from the colony
        for io in colony:
            for ip in io.rateOfUptakeOverTime:
                everyUptakeValue.append(ip)

        # sort the list into ascending numerical order
        everyUptakeValue.sort()

        # calculate the 10th Percentile

        # first calculate RANK of the 10th percentile
        rankOfTenthPercentileUptake = 0.1 * (float(len(everyUptakeValue)) + 1.0)

        # if calculation result is less than 0, then just use rank 1 (element [0] in the list)
        if rankOfTenthPercentileUptake < 1.0:
            rankOfTenthPercentileUptake = 0
            tenthPercentileUptake = float(everyUptakeValue[0])
        else:
        # if the calculated rank is a whole number
            if rankOfTenthPercentileUptake % int(rankOfTenthPercentileUptake) == 0.0:
                rankOfTenthPercentileUptake = int(rankOfTenthPercentileUptake)          # convert to type int and store this as the exact
rank
                tenthPercentileUptake = float(everyUptakeValue[(rankOfTenthPercentileUptake - 1)])      # look up tenth percentile
value (rank - 1 to factor in start from 0 in list)
            else:               # if there is a fraction part in the calculation output we need to interpolate
                tempIR = int(rankOfTenthPercentileUptake)                       # store the integer part of the calculated rank
                tempFR = rankOfTenthPercentileUptake % tempIR                       # store the fraction part of the calculated rank
                tempScoreIR = float(everyUptakeValue[(tempIR - 1)])                 # get the score with rank = integer part
                if len(everyUptakeValue) > tempIR:
                    tempScoreIRPlusOne = float(everyUptakeValue[(tempIR)])              # get the score with rank = integer part + 1
                    tenthPercentileUptake = (float(tempFR) * (tempScoreIRPlusOne - tempScoreIR)) + tempScoreIR      # calculate the
score
                else:
                    # if there is no higher rank, then just take the value of the highest rank in the list
                    tenthPercentileUptake = float(tempScoreIR)

        # calculate the 90th Percentile

        # first calculate RANK of the 90th percentile
        rankOfNinetiethPercentileUptake = 0.9 * (float(len(everyUptakeValue)) + 1.0)

        # if calculation result is less than 0, then just use rank 1 (element [0] in the list)
        if rankOfNinetiethPercentileUptake < 1.0:
            rankOfNinetiethPercentileUptake = 0
            ninetiethPercentileUptake = float(everyUptakeValue[0])
        else:
            # if the calculated rank is a whole number
            if rankOfNinetiethPercentileUptake % int(rankOfNinetiethPercentileUptake) == 0.0:
                rankOfNinetiethPercentileUptake = int(rankOfNinetiethPercentileUptake)          # convert to type int and store this as
the exact rank
                ninetiethPercentileUptake = float(everyUptakeValue[(rankOfNinetiethPercentileUptake - 1)])      # look up tenth
percentile value (rank - 1 to factor in start from 0 in list)
            else:               # if there is a fraction part in the calculation output we need to interpolate
                tempIR = int(rankOfNinetiethPercentileUptake)                       # store the integer part of the calculated rank
                tempFR = rankOfNinetiethPercentileUptake % tempIR                       # store the fraction part of the calculated rank
                tempScoreIR = float(everyUptakeValue[(tempIR - 1)])                 # get the score with rank = integer part
                if len(everyUptakeValue) > tempIR:
                    tempScoreIRPlusOne = float(everyUptakeValue[(tempIR)])              # get the score with rank = integer part + 1
                    ninetiethPercentileUptake = (float(tempFR) * (tempScoreIRPlusOne - tempScoreIR)) + tempScoreIR      # calculate
the score
                else:
                    # if there is no higher rank, then just take the value of the highest rank in the list
                    ninetiethPercentileUptake = float(tempScoreIR)

        # calculate mean of uptake rates
        meanUptakeRateAllValues = float(sum(everyUptakeValue)) / float(len(everyUptakeValue))

        #print "-In terms of Nectar Uptake Rates, the 10th Percentile is " + str(round(tenthPercentileUptake, 3))
        #print "-In terms of Nectar Uptake Rates, the 90th Percentile is " + str(round(ninetiethPercentileUptake, 3))
        #print "-In terms of Nectar Uptake Rates, the mean is " + str(round(meanUptakeRateAllValues, 3))
```

```
        # traffic over time per bee (on a bout by bout basis)
        figure(17)
        for iq in colony:
            plot(range(1, (len(iq.traffic) + 1)), iq.traffic)
        axis([1,passLifespan,0,50])
        xlabel('Foraging Bouts Completed')
        ylabel('Bout Traffic')
        title('Bout Traffic Per Bee Over Time')

        # sort the list of all traffic values in the colony into ascending numerical order
        colonyTraffic.sort()

        # calculate the 10th Percentile

        # first calculate RANK of the 10th percentile
        rankOfTenthPercentileTraffic = 0.1 * (float(len(colonyTraffic)) + 1.0)

        # if calculation result is less than 0, then just use rank 1 (element [0] in the list)
        if rankOfTenthPercentileTraffic < 1.0:
            rankOfTenthPercentileTraffic = 0
            tenthPercentileTraffic = float(colonyTraffic[0])
        else:
            # if the calculated rank is a whole number
            if rankOfTenthPercentileTraffic % int(rankOfTenthPercentileTraffic) == 0.0:
                rankOfTenthPercentileTraffic = int(rankOfTenthPercentileTraffic)          # convert to type int and store this as the exact
rank
                tenthPercentileTraffic = float(colonyTraffic[(rankOfTenthPercentileTraffic - 1)])      # look up tenth percentile value
(rank - 1 to factor in start from 0 in list)
            else:               # if there is a fraction part in the calculation output we need to interpolate
                tempIR = int(rankOfTenthPercentileTraffic)                      # store the integer part of the calculated rank
                tempFR = rankOfTenthPercentileTraffic % tempIR                  # store the fraction part of the calculated rank
                tempScoreIR = float(colonyTraffic[(tempIR - 1)])                # get the score with rank = integer part
                if len(colonyTraffic) > tempIR:
                    tempScoreIRPlusOne = float(colonyTraffic[(tempIR)])             # get the score with rank = integer part + 1
                    tenthPercentileTraffic = (float(tempFR) * (tempScoreIRPlusOne - tempScoreIR)) + tempScoreIR       # calculate the
score
                else:
                    # if there is no higher rank, then just take the value of the highest rank in the list
                    tenthPercentileTraffic = float(tempScoreIR)

        # calculate the 90th Percentile

        # first calculate RANK of the 90th percentile
        rankOfNinetiethPercentileTraffic = 0.9 * (float(len(colonyTraffic)) + 1.0)

        # if calculation result is less than 0, then just use rank 1 (element [0] in the list)
        if rankOfNinetiethPercentileTraffic < 1.0:
            rankOfNinetiethPercentileTraffic = 0
            ninetiethPercentileTraffic = float(colonyTraffic[0])
        else:
            # if the calculated rank is a whole number
            if rankOfNinetiethPercentileTraffic % int(rankOfNinetiethPercentileTraffic) == 0.0:
                rankOfNinetiethPercentileTraffic = int(rankOfNinetiethPercentileTraffic)          # convert to type int and store this as the
exact rank
                ninetiethPercentileTraffic = float(colonyTraffic[(rankOfNinetiethPercentileTraffic - 1)])      # look up tenth percentile
value (rank - 1 to factor in start from 0 in list)
            else:               # if there is a fraction part in the calculation output we need to interpolate
                tempIR = int(rankOfNinetiethPercentileTraffic)                      # store the integer part of the calculated rank
                tempFR = rankOfNinetiethPercentileTraffic % tempIR                  # store the fraction part of the calculated rank
                tempScoreIR = float(colonyTraffic[(tempIR - 1)])                # get the score with rank = integer part
                if len(colonyTraffic) > tempIR:
                    tempScoreIRPlusOne = float(colonyTraffic[(tempIR)])             # get the score with rank = integer part + 1
                    ninetiethPercentileTraffic = (float(tempFR) * (tempScoreIRPlusOne - tempScoreIR)) + tempScoreIR       # calculate
the score
                else:
                    # if there is no higher rank, then just take the value of the highest rank in the list
                    ninetiethPercentileTraffic = float(tempScoreIR)

        #print "-In terms of Bout Traffic Levels, the 10th Percentile is " + str(round(tenthPercentileTraffic, 3))
        #print "-In terms of Bout Traffic Levels, the 90th Percentile is " + str(round(ninetiethPercentileTraffic, 3))
        #print "-In terms of Bout Traffic Levels, the mean is " + str(round(meanColonyTraffic, 3))

        # number of full flowers per field over time
        figure(21)
        for dylIM in range(1, (dylNumberOfFieldsInLandscape + 1)):
            plot(range(1, (len(dylFullOverTime[dylIM]) + 1)), dylFullOverTime[dylIM], label = ('Field ' + str(dylIM)))
        axis([1,globalClock,0,dylStartingTotal[1]])
        xlabel('Time Elapsed in Simulation')
```

```
        ylabel('Number of FULL Flowers')
        legend()
        title('FULL Flowers Per Field Over Time')

        # build the dummy list for plotting EMPTY flower times
        for ia in timesWhenBeeSampledEMPTYFlowers:
          tempDummyList.append(0)

        figure(22)
        for ia in colony:
          if ia.beeID == allActionValuesBee:
            for dylIN in range(1, (dylNumberOfFieldsInLandscape + 1)):
              plot(range(0,len(ia.dylEstimatedQualityHistory[dylIN])), ia.dylEstimatedQualityHistory[dylIN], label = ('Field ' +
str(dylIN)))
            plot(timesWhenBeeSampledEMPTYFlowers, tempDummyList, 'ro')
        axis([0,globalClock,0.0,1.0])
        xlabel('Global Clock')
        ylabel('Estimated Quality')
        legend()
        title('Estimated Quality of Each Field For Bee ' + str(allActionValuesBee) + ' Over Time')

        # position of tracked bee within the landscape over time
        figure(25)
        for ie in colony:
          if ie.beeID == allActionValuesBee:
            plot(range(0,len(ie.allToursMerged)), ie.allToursMerged, 'ro')
        axis([0,globalClock,0,dylNumberOfFieldsInLandscape])
        xlabel('Global Clock')
        ylabel('Location of Bee')
        title('Location of BEE ' + str(allActionValuesBee) + ' Over Time')

        # remaining capacity of tracked bee over time
        figure(26)
        for ig in colony:
          if ig.beeID == allActionValuesBee:
            plot(range(0,len(ig.remainingCapacityHistory)), ig.remainingCapacityHistory)
        axis([0,globalClock,0,passfixedNectarCapacity])
        xlabel('Global Clock')
        ylabel('Remaining Capacity')
        title('Remaining Capacity of BEE ' + str(allActionValuesBee) + ' Over Time')

        if meanUptakeCalcOverTimeOnOff == 1:
          figure(27)
          for ih in colony:
            if ih.beeID == allActionValuesBee:
              for dylIO in range(1, (dylNumberOfFieldsInLandscape + 1)):
                plot(range(0,len(ih.dylMeanEstimatedQualityHistory[dylIO])), ih.dylMeanEstimatedQualityHistory[dylIO], label
= ('Field ' + str(dylIO)))
          axis([0,globalClock,0.0,1.0])
          xlabel('Global Clock')
          ylabel('Mean Estimated Quality')
          legend()
          title('Mean Estimated Quality of Each Field for BEE ' + str(allActionValuesBee) + ' Over Time')

        #print "...MEAN NUMBER OF BOUTS COMPLETED PER BEE = " +
str(round(meanBoutsCompletedThroughClock[(len(meanBoutsCompletedThroughClock) - 1)], 3))

        # calculate the TRIAL PERFORMANCE SCORE for each bee, then calculate the mean colony TRIAL PERFORMANCE
SCORE
        for ia in colony:
          tempTrialPerformance = float(((float(ia.foragingBoutTally) * float(ia.fixedNectarCapacity)) +
(float(ia.fixedNectarCapacity) - float(ia.remainingNectarCapacity)))) / (float(globalClock) / passSamplingInterval)
          allTrialPerformances.append(tempTrialPerformance)

        colonyTrialPerformance = float(sum(allTrialPerformances)) / float(len(colony))

        #print "...MEAN COLONY TRIAL PERFORMANCE = " + str(round(colonyTrialPerformance, 3))

        tempInt = int(sum(dylStartingTotal))

        figure(30)
        plot(range(0,len(landscapeStandingCropOverTime)), landscapeStandingCropOverTime)
        axis([0,globalClock,0,tempInt])
        xlabel('Global Clock')
        ylabel('FULL Flowers in Landscape')
        title('Total Standing Crop in Landscape Over Time')

        figure(31)
```

```
        plot(range(0,len(landscapeStandingCropOverTimeAveAcrossFields)), landscapeStandingCropOverTimeAveAcrossFields)
        axis([0,globalClock,0,tempInt])
        xlabel('Global Clock')
        ylabel('Average FULL Flowers')
        title('Average FULL Flowers Across Fields Over Time')

        #print "@@@ NEW FULL Flowers added to system  = " + str(newFlowersAddedToSystem)
        #print "@@@ FULL Flowers depleted from system = " + str(flowersDepletedFromSystem)
        ##print "-----"
        ##print "BEE " + str(allActionValuesBee) + " sampled EMPTY flowers at these times : "
        ##print timesWhenBeeSampledEMPTYFlowers

        # adds up the number of unique fields visited across the trial by each bee in the colony
        for dylIB in colony:
          dylNumberOfUniqueFieldsVisited.append(dylIB.dylFieldVisitedInLife.count(1))

        figure(32)
        plot(range(0,len(dylNumberOfUniqueFieldsVisited)), dylNumberOfUniqueFieldsVisited, 'ro')
        axis([0,len(dylNumberOfUniqueFieldsVisited),0,dylNumberOfFieldsInLandscape])
        xlabel('Bee ID')
        ylabel('Number Of Unique Fields Visited')
        title('Number Of Unique Fields Visited In Trial Per Bee')

        dylMeanNumberOfUniqueFieldsVisited = float(sum(dylNumberOfUniqueFieldsVisited)) /
float(len(dylNumberOfUniqueFieldsVisited))

        #print "...MEAN NUMBER OF UNIQUE FIELDS VISITED = " + str(dylMeanNumberOfUniqueFieldsVisited)

        figure(33)
        plot(range(0,len(dylRangeInStandingCropOverTime)), dylRangeInStandingCropOverTime)
        axis([0,len(dylRangeInStandingCropOverTime),0.0,1.0])
        xlabel('Global Clock')
        ylabel('Range in Field Qualities')
        title('Range in Landscape Field Qualities Over Time')

        #print "FULLs at Start : " + str(dylStartingFull)

        #print "Replenishment Rates : " + str(dylReplenishmentRate)

        show()

        clf
        cla

        raw_input("Press ENTER")
      else:
        resultsFile = open('resultsFile.txt', 'a')

        # if all bees completed at least one bout, then calculate mean colony traffic.
        # if not all bees completed at least one bout, then force an append of newest colony traffic figures for all bees that did not
        # complete at least one bout, and then calculate mean colony traffic as normal
        # (normally only completed bouts are counted for traffic)
        # ***NOTE - uses colony traffic to judge - might want to change as potentially bee completed 2 bouts could hide other bee
completed 0 bouts
        nonZeroBoutsByAllBees = 0

        for i in colony:
          if i.foragingBoutTally > 0:
            nonZeroBoutsByAllBees = nonZeroBoutsByAllBees + 1

        if nonZeroBoutsByAllBees >= len(colony):
          meanColonyTraffic = float(sum(colonyTraffic)) / float(len(colonyTraffic))
        else:
          for iz in colony:
            if iz.traffic == []:
              iz.traffic.append(iz.trafficCount)                    # append the traffic in the UNCOMPLETED AND ONLY bout for
the bee to the bee's total life traffic list
              colonyTraffic.append(iz.trafficCount)                 # also append to the colony's total traffic list

          meanColonyTraffic = float(sum(colonyTraffic)) / float(len(colonyTraffic))

        # calculate the TRIAL PERFORMANCE SCORE for each bee, then calculate the mean colony TRIAL PERFORMANCE
SCORE
        for ia in colony:
          tempTrialPerformance = float(((float(ia.foragingBoutTally) * float(ia.fixedNectarCapacity)) +
(float(ia.fixedNectarCapacity) - float(ia.remainingNectarCapacity)))) / (float(globalClock) / passSamplingInterval)
          allTrialPerformances.append(tempTrialPerformance)
```

```
        colonyTrialPerformance = float(sum(allTrialPerformances)) / float(len(colony))

        # find the UNLUCKIEST bee (one with lowest performance score) and grab its traffic, perf, bouts completed and unique fields
        unluckiestBee = allTrialPerformances.index(min(allTrialPerformances))

        for ib in colony:
            if ib.beeID == unluckiestBee:
                unluckiestTraffic = float(sum(ib.traffic)) / float(len(ib.traffic))
                unluckiestPerformance = float(min(allTrialPerformances))
                unluckiestBoutsCompleted = float(ib.foragingBoutTally)
                unluckiestUniqueFields = float(ib.dylFieldVisitedInLife.count(1))

        ##print "FULLs at Start : " + str(dylStartingFull)

        ##print "Replenishment Rates : " + str(dylReplenishmentRate)

        # adds up the number of unique fields visited across the trial by each bee in the colony
        for dylIB in colony:
            dylNumberOfUniqueFieldsVisited.append(dylIB.dylFieldVisitedInLife.count(1))

        dylMeanNumberOfUniqueFieldsVisited = float(sum(dylNumberOfUniqueFieldsVisited)) / float(len(dylNumberOfUniqueFieldsVisited))

        resultsFile.write(str(round(meanColonyTraffic, 3)) + "," + str(round(colonyTrialPerformance, 3)) + "," + str(round(meanBoutsCompletedThroughClock[(len(meanBoutsCompletedThroughClock) - 1)], 3)) + "," + str(dylMeanNumberOfUniqueFieldsVisited))
        resultsFile.write("\n")

        # write the mean foraging performance per bee to file
        meanPerformanceFile = open('meanPerformanceFile.txt','a')
        meanPerformanceFile.write(str(round(colonyTrialPerformance, 3)))
        meanPerformanceFile.write("\n")
        meanPerformanceFile.close()

        # write the mean number of foraging bouts completed to file
        meanBoutsCompletedFile = open('meanBoutsCompletedFile.txt','a')
        meanBoutsCompletedFile.write(str(round(meanBoutsCompletedThroughClock[(len(meanBoutsCompletedThroughClock) - 1)],3)))
        meanBoutsCompletedFile.write("\n")
        meanBoutsCompletedFile.close()

        # write the mean number of unique patches visited value to the meanUniquePatchesVisited file
        meanUniquePatchesVisited = open('meanUniquePatchesVisited.txt','a')
        meanUniquePatchesVisited.write(str(round(dylMeanNumberOfUniqueFieldsVisited, 3)))
        meanUniquePatchesVisited.write("\n")
        meanUniquePatchesVisited.close()

        # write the mean traffic value to the meanTrafficFile as well
        meanTrafficFile = open('meanTrafficFile.txt','a')
        meanTrafficFile.write(str(round(meanColonyTraffic, 3)))
        meanTrafficFile.write("\n")
        meanTrafficFile.close()

        # work out the mean total traffic in trial per bee and write to file
        tempSumTotalTraffic = 0.0
        tempTotalTrafficForEachBee = []

        for i in colony:
            for j in range(0, len(i.traffic)):
                tempSumTotalTraffic = float(tempSumTotalTraffic) + float(i.traffic[j])
            tempTotalTrafficForEachBee.append(float(tempSumTotalTraffic))
            tempSumTotalTraffic = 0.0

        tempSumTotalTraffic = 0.0

        for i in range(0, len(tempTotalTrafficForEachBee)):
            tempSumTotalTraffic = float(tempSumTotalTraffic) + float(tempTotalTrafficForEachBee[i])
        tempMeanTotalTraffic = float(tempSumTotalTraffic) / float(len(tempTotalTrafficForEachBee))

        meanTOTALTrafficFile = open('meanTOTALTrafficFile.txt','a')
        meanTOTALTrafficFile.write(str(round(tempMeanTotalTraffic,3)))
        meanTOTALTrafficFile.write("\n")
        meanTOTALTrafficFile.close()

        # write unluckiest bee data to file
        resultsFile.write("---UNLUCKIEST BEE DATA---")
        resultsFile.write("\n")
```

```
        resultsFile.write(str(round(unluckiestTraffic, 3)) + "," + str(round(unluckiestPerformance, 3)) + "," + str(round(unluckiestBoutsCompleted, 3)) + "," + str(round(unluckiestUniqueFields, 3)))
        resultsFile.write("\n")

        # convert the transition matrix cells to proportions (so each cell shows (x,y) transitions as proportion of total transitions)
        # NEST NOT INCLUDED
        for ie in range(1, len(transitionMatrix)):
            runningSumTransitionMatrix = runningSumTransitionMatrix + (sum(transitionMatrix[ie]) - transitionMatrix[ie][0])

        totalOfTransitionMatrix = runningSumTransitionMatrix

        for ig in range(1, len(transitionMatrix)):
            for ih in range(1, len(transitionMatrix[ig])):
                transitionMatrix[ig][ih] = float(transitionMatrix[ig][ih]) / float(totalOfTransitionMatrix)

        # if more than 1 bee, calculate bee-level standard errors for traffic, perf, bouts completed and unique fields

        if maximumInjections > 1:
            for iz in colony:
                squaredDifferenceTraffic.append(float(pow(((float(sum(iz.traffic)) / float(len(iz.traffic))) - (float(meanColonyTraffic))), 2.0)))
                squaredDifferencePerformance.append(float(pow((float(allTrialPerformances[iz.beeID]) - float(colonyTrialPerformance)),2.0)))
                squaredDifferenceBoutsCompleted.append(float(pow((float(iz.foragingBoutTally) - float(meanBoutsCompletedThroughClock[(len(meanBoutsCompletedThroughClock) - 1)])), 2.0)))
                squaredDifferenceUniqueFields.append(float(pow((float(iz.dylFieldVisitedInLife.count(1)) - float(dylMeanNumberOfUniqueFieldsVisited)), 2.0)))

            varianceTraffic = float(sum(squaredDifferenceTraffic)) / float((float(maximumInjections) - 1.0))
            variancePerformance = float(sum(squaredDifferencePerformance)) / float((float(maximumInjections) - 1.0))
            varianceBoutsCompleted = float(sum(squaredDifferenceBoutsCompleted)) / float((float(maximumInjections) - 1.0))
            varianceUniqueFields = float(sum(squaredDifferenceUniqueFields)) / float((float(maximumInjections) - 1.0))

            standardDeviationTraffic = float(math.sqrt(varianceTraffic))
            standardDeviationPerformance = float(math.sqrt(variancePerformance))
            standardDeviationBoutsCompleted = float(math.sqrt(varianceBoutsCompleted))
            standardDeviationUniqueFields = float(math.sqrt(varianceUniqueFields))

            standardErrorTraffic = float(standardDeviationTraffic) / float(math.sqrt(maximumInjections))
            standardErrorPerformance = float(standardDeviationPerformance) / float(math.sqrt(maximumInjections))
            standardErrorBoutsCompleted = float(standardDeviationBoutsCompleted) / float(math.sqrt(maximumInjections))
            standardErrorUniqueFields = float(standardDeviationUniqueFields) / float(math.sqrt(maximumInjections))

            resultsFile.write("---STANDARD ERRORS---")
            resultsFile.write("\n")
            resultsFile.write(str(standardErrorTraffic) + "," + str(standardErrorPerformance) + "," + str(standardErrorBoutsCompleted) + "," + str(standardErrorUniqueFields))
            resultsFile.write("\n")

        # write the converted transition matrix to the results file
        resultsFile.write("---TRANSITION MATRIX---")
        resultsFile.write("\n")
        for ii in range(0, len(transitionMatrix)):
            for ij in range(0, len(transitionMatrix[ii])):
                resultsFile.write(str(transitionMatrix[ii][ij]))
                resultsFile.write(",")
            resultsFile.write("\n")

        # also write to the transitionMatrixFile
        transitionMatrixFile = open('transitionMatrixFile.txt','a')
        for ii in range(0, len(transitionMatrix)):
            for ij in range(0, len(transitionMatrix[ii])):
                transitionMatrixFile.write(str(transitionMatrix[ii][ij]))
                transitionMatrixFile.write(",")
            transitionMatrixFile.write("\n")
        transitionMatrixFile.write("\n")
        transitionMatrixFile.close()

        # calculate gene flow, based on GM field = field with highest rate of emigration (ie - worst case scenario (maximum gene flow))
        highestEmigrationField = 1                  # set to 1 by default

        for ix in range(1, len(transitionMatrix)):
            tempGeneSum = sum(transitionMatrix[ix]) - float(transitionMatrix[ix][0]) - float(transitionMatrix[ix][ix])
            if tempGeneSum > tempPreviousGeneSum:
                tempPreviousGeneSum = tempGeneSum
                tempGeneSum = 0.0
                highestEmigrationField = ix
```

```
        geneFlowE = sum(transitionMatrix[highestEmigrationField]) - float(transitionMatrix[highestEmigrationField][0]) - float(transitionMatrix[highestEmigrationField][highestEmigrationField])

        tempGeneSum = 0.0

        for iy in range(1, len(transitionMatrix)):
            tempGeneSum = float(tempGeneSum) + float(transitionMatrix[iy][iy])

        tempGeneSum = float(tempGeneSum) - float(transitionMatrix[highestEmigrationField][highestEmigrationField])

        averagePercentStaysNonGM = float(tempGeneSum) / float(len(transitionMatrix) - 2.0)

        geneFlowb = float(averagePercentStaysNonGM) * (float(passfixedNectarCapacity) / float(proposedMeanLandscapeQuality))

        if geneFlowb == 0.0:
            calculatedGeneFlow = 0.0
        else:
            calculatedGeneFlow = float(geneFlowE) * (float(geneFlowFF) / float(geneFlowb))

        # !!! IMPORTANT - below method only works for NO DEP / REP and IDENTICAL TOTAL FLOWERS PER FIELD
        # build the list which is to be a sorted version of dylStartingFull
        for ia in dylStartingFull:
            startingFullSorted.append(ia)

        # sort the list
        startingFullSorted.sort()

        # reverse the list
        startingFullSorted.reverse()

        # find the FULLs (at start) of the highest emigration field
        fullsOfHighestEmigrationField = dylStartingFull[highestEmigrationField]

        # work out ranking of highest emigration field (in terms of field quality) ('+ 1' to offset 0 start of array)
        rankingOfHighestEmigrationField = startingFullSorted.index(fullsOfHighestEmigrationField) + 1

        resultsFile.write("---GENE FLOW DETAILS--- (Highest Em Field, Calc Gene Flow, Ranking Highest Em Field, E, b)")
        resultsFile.write("\n")
        resultsFile.write(str(highestEmigrationField) + "," + str(calculatedGeneFlow) + "," + str(rankingOfHighestEmigrationField) + "," + str(geneFlowE) + "," + str(geneFlowb))
        resultsFile.write("\n")

        # write the starting FULL flowers from each field to the file
        resultsFile.write("---STARTING RESOURCES---")
        resultsFile.write("\n")
        for ik in range(0, len(dylStartingFull)):
            resultsFile.write(str(dylStartingFull[ik]))
            resultsFile.write(",")

        resultsFile.write("\n")
        resultsFile.write("\n")

        resultsFile.close()

        # write the residence per patch for each bee to the residenceResultsRAW.txt file
        residenceResultsRAW = open('residenceResultsRAW.txt','w')

        # for each bee in the colony
        for ia in colony:
            # write header for this bee
            residenceResultsRAW.write("Bee ")
            residenceResultsRAW.write(str(ia.beeID))
            residenceResultsRAW.write("\n")
            residenceResultsRAW.write("------")
            residenceResultsRAW.write("\n")
            residenceResultsRAW.write("Format : 1)Patch Number 2)Patch Type 3)Residence (Flower Visits)")
            residenceResultsRAW.write("\n")

            # write all of the residence entries for this bee
            for ib in range(0, len(ia.residencePerPatch)):
                residenceResultsRAW.write(str(ia.residencePerPatch[ib][1]))
                residenceResultsRAW.write(",")
                residenceResultsRAW.write(str(ia.residencePerPatch[ib][2]))
                residenceResultsRAW.write(",")
                residenceResultsRAW.write(str(ia.residencePerPatch[ib][0]))
                residenceResultsRAW.write("\n")
```

```
            residenceResultsRAW.write("\n")

        residenceResultsRAW.close()

        # now, work out the average residences (per patch for each patch type, and per patch for all patch types) for each bee and write to file
        # for each bee in the colony
        residenceResultsAVERAGE = open('residenceResultsAVERAGE.txt','a')

        residenceResultsAVERAGE.write("Format : 1)Type of Average 2)Average Value 3)Number of Visits to this type")
        residenceResultsAVERAGE.write("\n")
        residenceResultsAVERAGE.write("--------------------------------------------------------------------------------")
        residenceResultsAVERAGE.write("\n")

        # set up the correct number of empty sub-lists in the specificResidenceLog
        for i in range(0, len(typesOfPatch)):
            specificResidenceLog.append([])

        # set up the correct number of empty sub-lists in the eachResidenceLog
        for i in range(0, (dylNumberOfFieldsInLandscape + 1)):
            eachResidenceLog.append([])

        for ic in colony:
            residenceResultsAVERAGE.write("Bee ")
            residenceResultsAVERAGE.write(str(ic.beeID))
            residenceResultsAVERAGE.write("\n")
            residenceResultsAVERAGE.write("------")
            residenceResultsAVERAGE.write("\n")

            tempSumPersonalResidence = 0                # reset to 0
            tempMeanPersonalResidence = 0.0             # reset to 0.0

            # first, work out the average residence per patch across ALL patch types
            # add up all of the residences
            for ie in range(0, len(ic.residencePerPatch)):
                tempSumPersonalResidence = tempSumPersonalResidence + ic.residencePerPatch[ie][0]

            # take the mean
            if len(ic.residencePerPatch) != 0:
                tempMeanPersonalResidence = float(tempSumPersonalResidence) / float(len(ic.residencePerPatch))
            else:
                tempMeanPersonalResidence = 0.0

            # write the mean to file
            residenceResultsAVERAGE.write("ALL,")
            residenceResultsAVERAGE.write(str(tempMeanPersonalResidence))
            residenceResultsAVERAGE.write(",")
            residenceResultsAVERAGE.write(str(len(ic.residencePerPatch)))
            residenceResultsAVERAGE.write("\n")

            # append the mean to the allResidenceLog
            allResidenceLog.append(float(tempMeanPersonalResidence))

            tempSumPersonalResidence = 0                # reset to 0
            tempMeanPersonalResidence = 0.0             # reset to 0.0

            # second, work out the average residence per patch for each patch type SEPARATELY
            # for each patch type
            for ig in range(0, len(typesOfPatch)):
                tempSumPersonalResidence = 0            # reset to 0
                tempMeanPersonalResidence = 0.0         # reset to 0.0
                tempInt = 0

                # add up all of the residences where patch type = this patch type
                for ih in range(0, len(ic.residencePerPatch)):
                    if ic.residencePerPatch[ih][2] == typesOfPatch[ig]:
                        tempSumPersonalResidence = tempSumPersonalResidence + ic.residencePerPatch[ih][0]
                        tempInt = tempInt + 1           # increment the counter by 1 (for calculating the mean)

                # take the mean
                if tempInt != 0:
                    tempMeanPersonalResidence = float(tempSumPersonalResidence) / float(tempInt)
                else:
                    tempMeanPersonalResidence = 0.0

                # write the mean to file
                residenceResultsAVERAGE.write(str(typesOfPatch[ig]))
```

```
            residenceResultsAVERAGE.write(",")
            residenceResultsAVERAGE.write(str(tempMeanPersonalResidence))
            residenceResultsAVERAGE.write(",")
            residenceResultsAVERAGE.write(str(tempInt))
            residenceResultsAVERAGE.write("\n")

            # append the mean to the relevant sublist of the specificResidenceLog
            specificResidenceLog[ig].append(tempMeanPersonalResidence)

        residenceResultsAVERAGE.write("\n")

        # now work out the average residence per patch for each patch separately
        for i in range(1, (dylNumberOfFieldsInLandscape + 1)):
            tempSumPersonalResidence = 0                # reset to 0
            tempMeanPersonalResidence = 0.0             # reset to 0.0
            tempInt = 0

            # add up all of the residences where patch = this patch
            for ih in range(0, len(ic.residencePerPatch)):
                if ic.residencePerPatch[ih][1] == i:
                    tempSumPersonalResidence = tempSumPersonalResidence + ic.residencePerPatch[ih][0]
                    tempInt = tempInt + 1               # increment the counter by 1 (for calculating the mean)

            # take the mean
            if tempInt != 0:
                tempMeanPersonalResidence = float(tempSumPersonalResidence) / float(tempInt)
            else:
                tempMeanPersonalResidence = 0.0

            # append the mean to the relevant sublist of the eachResidenceLog
            eachResidenceLog[i].append(tempMeanPersonalResidence)

    residenceResultsAVERAGE.close()

    # add the total visits to ALL patches for each bee to ALLVisitsLog
    for i in colony:
        ALLVisitsLog.append(len(i.residencePerPatch))

    # work out mean
    tempMeanAllVisits = 0.0

    for i in range(0, len(ALLVisitsLog)):
        tempMeanAllVisits = float(tempMeanAllVisits) + float(ALLVisitsLog[i])

    tempMeanAllVisits = float(tempMeanAllVisits) / float(len(ALLVisitsLog))

    # write to file
    meanALLVisitsFile = open('meanALLVisitsFile.txt','a')
    meanALLVisitsFile.write(str(round(tempMeanAllVisits, 3)))
    meanALLVisitsFile.write("\n")
    meanALLVisitsFile.close()

    # add the total visits to POOR patches for each bee to POORVisitsLog
    for i in colony:
        tempInt = 0

        # add up all of the visits where patch type = Poor
        for j in range(0, len(i.residencePerPatch)):
            if i.residencePerPatch[j][2] == "Poor":
                tempInt = tempInt + 1               # increment the counter by 1

        POORVisitsLog.append(tempInt)

    # work out mean
    tempMeanPoorVisits = 0.0

    for i in range(0, len(POORVisitsLog)):
        tempMeanPoorVisits = float(tempMeanPoorVisits) + float(POORVisitsLog[i])

    tempMeanPoorVisits = float(tempMeanPoorVisits) / float(len(POORVisitsLog))

    # write to file
    meanPOORVisitsFile = open('meanPOORVisitsFile.txt','a')
    meanPOORVisitsFile.write(str(round(tempMeanPoorVisits, 3)))
    meanPOORVisitsFile.write("\n")
    meanPOORVisitsFile.close()

    # add the total visits to GOOD patches for each bee to GOODVisitsLog
```

```
for i in colony:
    tempInt = 0

    # add up all of the visits where patch type = Good
    for j in range(0, len(i.residencePerPatch)):
        if i.residencePerPatch[j][2] == "Good":
            tempInt = tempInt + 1                # increment the counter by 1

    GOODVisitsLog.append(tempInt)

# work out mean
tempMeanGoodVisits = 0.0

for i in range(0, len(GOODVisitsLog)):
    tempMeanGoodVisits = float(tempMeanGoodVisits) + float(GOODVisitsLog[i])

tempMeanGoodVisits = float(tempMeanGoodVisits) / float(len(GOODVisitsLog))

# write to file
meanGOODVisitsFile = open('meanGOODVisitsFile.txt','a')
meanGOODVisitsFile.write(str(round(tempMeanGoodVisits, 3)))
meanGOODVisitsFile.write("\n")
meanGOODVisitsFile.close()

# add the total visits to each patch for each bee to EACHVisitsLog
# append empty list to EACHVisitsLog for each patch in the system
for i in range(0, (dylNumberOfFieldsInLandscape + 1)):
    EACHVisitsLog.append([])

for k in range(1, (dylNumberOfFieldsInLandscape + 1)):
    for i in colony:
        tempInt = 0

        # add up all of the visits where patch = this patch
        for j in range(0, len(i.residencePerPatch)):
            if i.residencePerPatch[j][1] == k:
                tempInt = tempInt + 1                # increment the counter by 1

        EACHVisitsLog[k].append(tempInt)

# work out means and write to file
meanEACHVisitsFile = open('meanEACHVisitsFile.txt','a')

for k in range(1, (dylNumberOfFieldsInLandscape + 1)):
    tempMeanEachVisits = 0.0

    for i in range(0, len(EACHVisitsLog[k])):
        tempMeanEachVisits = float(tempMeanEachVisits) + float(EACHVisitsLog[k][i])

    tempMeanEachVisits = float(tempMeanEachVisits) / float(len(EACHVisitsLog[k]))

    # write to file
    meanEACHVisitsFile.write(str(round(tempMeanEachVisits, 3)))
    meanEACHVisitsFile.write(",")

meanEACHVisitsFile.write("\n")
meanEACHVisitsFile.close()

# work out the average residences across the colony and write to the meanResidenceFiles
meanResidenceFileALL = open('meanResidenceFileALL.txt','a')
meanResidenceFilePOOR = open('meanResidenceFilePOOR.txt','a')
meanResidenceFileGOOD = open('meanResidenceFileGOOD.txt','a')
meanResidenceFileEACH = open('meanResidenceFileEACH.txt','a')

tempMeanPersonalResidence = float(sum(allResidenceLog)) / float(len(allResidenceLog))
meanResidenceFileALL.write(str(round(tempMeanPersonalResidence,3)))
meanResidenceFileALL.write("\n")

tempMeanPersonalResidence = float(sum(specificResidenceLog[0])) / float(len(specificResidenceLog[0]))

meanResidenceFilePOOR.write(str(round(tempMeanPersonalResidence,3)))
meanResidenceFilePOOR.write("\n")

tempMeanPersonalResidence = float(sum(specificResidenceLog[1])) / float(len(specificResidenceLog[1]))

meanResidenceFileGOOD.write(str(round(tempMeanPersonalResidence,3)))
meanResidenceFileGOOD.write("\n")
```

```
        for i in range(1, len(eachResidenceLog)):
            tempMeanPersonalResidence = float(sum(eachResidenceLog[i])) / float(len(eachResidenceLog[i]))
            meanResidenceFileEACH.write(str(round(tempMeanPersonalResidence,3)))
            meanResidenceFileEACH.write(",")

        meanResidenceFileEACH.write("\n")

        meanResidenceFileALL.close()
        meanResidenceFilePOOR.close()
        meanResidenceFileGOOD.close()
        meanResidenceFileEACH.close()

        # write the consecutive list elements to the graph 1 results file (cumulative unique fields VS time)
        #graph1Results = open('graph1Results.txt', 'a')

        #for ia,ib in enumerate(listOfMeanUniques):
            #graph1Results.write(str(listOfMeanUniques[ia]))
            #graph1Results.write("\n")

        #graph1Results.write("\n")

        #graph1Results.close()

        # write the consecutive list elements to the graph 2 results file (foraging performance vs time)
        #graph2Results = open('graph2Results.txt', 'a')

        #for ic,ie in enumerate(listOfMeanPerformances):
            #graph2Results.write(str(listOfMeanPerformances[ic]))
            #graph2Results.write("\n")

        #graph2Results.write("\n")

        #graph2Results.close()

        # grab listOfUniquesPerBout from each forager to form meta list of lists
        #for ic in colony:
          # metaListOfUniquesPerBout.append(ic.listOfUniquesPerBout)

        # create a new list of lengths of each sub-list in the meta list
        #for ia,ib in enumerate(metaListOfUniquesPerBout):
            #LENGTHSmetaListOfUniquesPerBout.append(len(metaListOfUniquesPerBout[ia]))

        # find the shortest list length from the meta list - this determines the number of bouts we will analyse for unique field data
        #lengthOfShortestUniqueList = min(LENGTHSmetaListOfUniquesPerBout)

        # for each bout we are analysing, take the mean unique fields visited across the colony and add to the main list
        #for ie in range(0,lengthOfShortestUniqueList):
            #for ig in colony:
                #tempBoutUniquesSum = float(tempBoutUniquesSum) + float(ig.listOfUniquesPerBout[ie])

            #tempMeanBoutUniques = float(tempBoutUniquesSum) / float(len(colony))

            #listOfMeanBoutUniques.append(tempMeanBoutUniques)

            # reset parameters
            #tempBoutUniquesSum = 0.0
            #tempMeanBoutUniques = 0.0

        # write the consecutive list elements to the graph 3 results file (unique fields per bout VS bout)
        #graph3Results = open('graph3Results.txt', 'a')

        #for ih,ii in enumerate(listOfMeanBoutUniques):
            #graph3Results.write(str(listOfMeanBoutUniques[ih]))
            #graph3Results.write("\n")

        #graph3Results.write("\n")

        #graph3Results.close()

def startProgram():
    batchTempRead = ''                          # temporarily stores non-value lines read-in from the batch parameters file
    batchNumberOfRuns = 1                       # stores the number of times the simulation should run
    firstLookAtColonySize = 0                   # stores the size of the colony

    trialSetCode = 0                            # logs the code of this trial set for reference purposes - CHANGE HERE

    # list of omega values to apply
    dOmega = [-1]
```

```
    #dOmega                                                                                                            =
[0.0,0.02,0.04,0.06,0.08,0.1,0.12,0.14,0.16,0.18,0.2,0.22,0.24,0.26,0.28,0.3,0.32,0.34,0.36,0.38,0.4,0.42,0.44,0.46,0.48,0.5,0.52,0.54
,0.56,0.58,0.6,0.62,0.64,0.66,0.68,0.7,0.72,0.74,0.76,0.78,0.8,0.82,0.84,0.86,0.88,0.9,0.92,0.94,0.96,0.98,1.0]

    batchParametersFile = open ('batchParametersFile.txt', 'r')        # opens file with user defined batch parameters

    # read in parameters for the batch operations from the file
    batchTempRead = batchParametersFile.readline()
    batchNumberOfRuns = int(batchParametersFile.readline())

    batchParametersFile.close()

    print "! Simulation Started at " + str(time.ctime(time.time())) + " !"

    resultsFile = open('resultsFile.txt', 'a')      # open for appending

    resultsFile.write("******** TRIAL SET CODE = " + str(trialSetCode) + " ********")
    resultsFile.write("\n")
    resultsFile.write("----------------------")
    resultsFile.write("\n")

    resultsFile.close()

    # read in the size of the colony
    globalParametersFile = open('globalParametersFile.txt','r')
    globalParametersFile.readline()
    globalParametersFile.readline()
    globalParametersFile.readline()
    firstLookAtColonySize = int(globalParametersFile.readline())
    globalParametersFile.close()

    for ig in range(0, firstLookAtColonySize):
        beeXTour = open('tourFiles/bee' + str(ig) + 'Tour.txt', 'w')
        beeXTour.write("TOUR OF BEE " + str(ig))
        beeXTour.write("\n")
        beeXTour.write("----------------")
        beeXTour.write("\n")
        beeXTour.close()

    # open file for writing to wipe previous trial data, then immediately close
    patchQualitiesOverTime = open('GraphData/patchQualitiesOverTime.txt','w')
    patchQualitiesOverTime.close()

    # read in the initial values in the global parameters file so they can be set back to this when the file is re-written on each run
    #globalParametersFile = open('globalParametersFile.txt','r')

    #globalParametersFile.readline()
    #originalInjectionInterval = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalMaximumInjections = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalTimeToCompleteAtNest = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalSimulationCutOff = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalActionValueTrackingOnOff = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalActionValueTrackingBee = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalDepletionConstant = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalMeanUptake = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalSuddenResourceInjectionOnOff = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalSuddenResourceInjectionAmount = int(globalParametersFile.readline())

    #globalParametersFile.readline()
    #originalSuddenResourceInjectionTime = int(globalParametersFile.readline())
```

```
#globalParametersFile.readline()
#originalSuddenResourceInjectionField = int(globalParametersFile.readline())

#globalParametersFile.readline()
#originalOmniscience = int(globalParametersFile.readline())

#globalParametersFile.readline()
#originalPauseToCheckParameters = int(globalParametersFile.readline())

#globalParametersFile.readline()
#originalMinimalLog = int(globalParametersFile.readline())

#globalParametersFile.readline()
#originalGamma = float(globalParametersFile.readline())

#globalParametersFile.readline()
#originalTrivialDistanceNest = int(globalParametersFile.readline())

#globalParametersFile.readline()
#globalParametersFile.readline()

#globalParametersFile.close()

# repeat for each value of omega to be investigated
for batchOuterCounter in range(0,len(dOmega)):
   #print "Omega = " + str(dOmega[batchOuterCounter])

   # re-write the global parameters file with the correct omega value
   #globalParametersFile = open('globalParametersFile.txt','w')

   #globalParametersFile.write("Bee Injection Interval:")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalInjectionInterval))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("Maximum Injections:")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalMaximumInjections))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("Time to Complete AT NEST:")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalTimeToCompleteAtNest))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("Simulation Cut-Off:")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalSimulationCutOff))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("Action Value Tracking On / Off:")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalActionValueTrackingOnOff))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("Action Value / Est Qual / Location / Rem Cap Tracking Bee :")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalActionValueTrackingBee))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("DEPLETION Constant:")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalDepletionConstant))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("Mean Uptake Over Time Calculation On / Off:")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalMeanUptake))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("Sudden Resource Injection On / Off:")
   #globalParametersFile.write("\n")
   #globalParametersFile.write(str(originalSuddenResourceInjectionOnOff))
   #globalParametersFile.write("\n")

   #globalParametersFile.write("Sudden Resource Injection Amount:")
   #globalParametersFile.write("\n")
```

```
#globalParametersFile.write(str(originalSuddenResourceInjectionAmount))
#globalParametersFile.write("\n")

#globalParametersFile.write("Sudden Resource Injection Time:")
#globalParametersFile.write("\n")
#globalParametersFile.write(str(originalSuddenResourceInjectionTime))
#globalParametersFile.write("\n")

#globalParametersFile.write("Sudden Resource Injection Field (A = 0, B = 1, C = 2):")
#globalParametersFile.write("\n")
#globalParametersFile.write(str(originalSuddenResourceInjectionField))
#globalParametersFile.write("\n")

#globalParametersFile.write("Colony Omniscience (0 = OFF, 1 = ON):")
#globalParametersFile.write("\n")
#globalParametersFile.write(str(originalOmniscience))
#globalParametersFile.write("\n")

#globalParametersFile.write("PAUSE to Check Parameters:")
#globalParametersFile.write("\n")
#globalParametersFile.write(str(originalPauseToCheckParameters))
#globalParametersFile.write("\n")

#globalParametersFile.write("Minimal Log:")
#globalParametersFile.write("\n")
#globalParametersFile.write(str(originalMinimalLog))
#globalParametersFile.write("\n")

#globalParametersFile.write("Gamma:")
#globalParametersFile.write("\n")
#globalParametersFile.write(str(originalGamma))
#globalParametersFile.write("\n")

#globalParametersFile.write("Trivial-Distance Nest (0 = OFF, 1 = ON)")
#globalParametersFile.write("\n")
#globalParametersFile.write(str(originalTrivialDistanceNest))
#globalParametersFile.write("\n")

#globalParametersFile.write("Omega:")
#globalParametersFile.write("\n")
#globalParametersFile.write(str(dOmega[batchOuterCounter]))
#globalParametersFile.write("\n")

#globalParametersFile.close()

# create a header for this omega value in the results file
#resultsFile = open('resultsFile.txt','a')      # open for appending
#resultsFile.write("Omega = " + str(dOmega[batchOuterCounter]))
#resultsFile.write("\n")
#resultsFile.write("----------")
#resultsFile.write("\n")
#resultsFile.close()

# create a header for this omega value in the mean traffic file
#meanTrafficFile = open('meanTrafficFile.txt','a')      # open for appending
#meanTrafficFile.write("Omega = " + str(dOmega[batchOuterCounter]))
#meanTrafficFile.write("\n")
#meanTrafficFile.write("----------")
#meanTrafficFile.write("\n")
#meanTrafficFile.close()

# create a header for this omega value in the mean residence file
#meanResidenceFile = open('meanResidenceFile.txt','a')      # open for appending
#meanResidenceFile.write("Omega = " + str(dOmega[batchOuterCounter]))
#meanResidenceFile.write("\n")
#meanResidenceFile.write("----------")
#meanResidenceFile.write("\n")
#meanResidenceFile.close()

# empty the results file to start anew
resultsFile = open('resultsFile.txt','w')
resultsFile.close()

# empty the residenceResultsAVERAGE file to start anew
residenceResultsAVERAGE = open('residenceResultsAVERAGE.txt','w')
residenceResultsAVERAGE.close()

# empty the mean traffic file to start anew
```

```
        meanTrafficFile = open('meanTrafficFile.txt','w')
        meanTrafficFile.close()

        # empty the mean total traffic file to start anew
        meanTOTALTrafficFile = open('meanTOTALTrafficFile.txt','w')
        meanTOTALTrafficFile.close()

        # empty the meanResidenceFileALL to start anew
        meanResidenceFileALL = open('meanResidenceFileALL.txt','w')
        meanResidenceFileALL.close()

        # empty the meanResidenceFilePOOR to start anew
        meanResidenceFilePOOR = open('meanResidenceFilePOOR.txt','w')
        meanResidenceFilePOOR.close()

        # empty the meanResidenceFileGOOD to start anew
        meanResidenceFileGOOD = open('meanResidenceFileGOOD.txt','w')
        meanResidenceFileGOOD.close()

        # empty the meanResidenceFileEACH to start anew
        meanResidenceFileEACH = open('meanResidenceFileEACH.txt','w')
        meanResidenceFileEACH.close()

        # empty the meanUniquePatchesVisited file to start anew
        meanUniquePatchesVisited = open('meanUniquePatchesVisited.txt','w')
        meanUniquePatchesVisited.close()

        # empty the meanBoutsCompletedFile file to start anew
        meanBoutsCompletedFile = open('meanBoutsCompletedFile.txt','w')
        meanBoutsCompletedFile.close()

        # empty the meanPerformanceFile file to start anew
        meanPerformanceFile = open('meanPerformanceFile.txt','w')
        meanPerformanceFile.close()

        # empty the meanALLVisitsFile file to start anew
        meanALLVisitsFile = open('meanALLVisitsFile.txt','w')
        meanALLVisitsFile.close()

        # empty the meanPOORVisitsFile file to start anew
        meanPOORVisitsFile = open('meanPOORVisitsFile.txt','w')
        meanPOORVisitsFile.close()

        # empty the meanGOODVisitsFile file to start anew
        meanGOODVisitsFile = open('meanGOODVisitsFile.txt','w')
        meanGOODVisitsFile.close()

        # empty the meanEACHVisitsFile file to start anew
        meanEACHVisitsFile = open('meanEACHVisitsFile.txt','w')
        meanEACHVisitsFile.close()

        # empty the transitionMatrixFile file to start anew
        transitionMatrixFile = open('transitionMatrixFile.txt','w')
        transitionMatrixFile.close()

        # empty the naivety report to start anew
        naivetyReport = open('naivetyReport.txt','w')
        naivetyReport.close()

        # repeat the simulation the number of times specified
        for batchSubCounter in range(0,batchNumberOfRuns):
            print ">>>BatchSubCounter -- " + str(batchSubCounter)

            # set up the naivety report header for this trial
            naivetyReport = open('naivetyReport.txt','a')

            naivetyReport.write("***TRIAL " + str(batchSubCounter) + " Naivety Report.  Format=bee ID, time on global clock bee
became naive.")
            naivetyReport.write("\n")

            naivetyReport.close()

            main = Main()
            passThisNest = Nest()
            passThisNest.initialSetup()
            main.runMultipleBees(passThisNest)

def startGridPlayer():
```

```
    gridPlayer = GridPlayer()
    gridPlayer.playTour()
```

LAUNCHER.py

```
from Tkinter import *
import threading
import math

class AutoScrollbar(Scrollbar):
    # a scrollbar that hides itself if it's not needed.  only
    # works if you use the grid geometry manager.
    def set(self, lo, hi):
        if float(lo) <= 0.0 and float(hi) >= 1.0:
            # grid_remove is currently missing from Tkinter!
            self.tk.call("grid", "remove", self)
        else:
            self.grid()
        Scrollbar.set(self, lo, hi)
    def pack(self, **kw):
        raise TclError, "cannot use pack with this widget"
    def place(self, **kw):
        raise TclError, "cannot use place with this widget"

class GUIInput(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.master.title("HARVEST (Harvesting Animal Reinforced Values and ESTimates)")

        content1 = StringVar()
        content2 = StringVar()
        content3 = StringVar()
        content4 = StringVar()
        content5 = StringVar()
        content6 = StringVar()
        content7 = StringVar()
        content8 = StringVar()
        content9 = StringVar()
        content10 = StringVar()
        content13 = StringVar()
        content14 = StringVar()
        content15 = StringVar()
        content16 = StringVar()
        content17 = StringVar()
        content18 = StringVar()
        content19 = StringVar()
        content22 = StringVar()
        content23 = StringVar()
        content24 = StringVar()
        content25 = StringVar()
        content26 = StringVar()
        content27 = StringVar()
        contentPPSizes = StringVar()
        contentPCoords = StringVar()
        launchStatus = StringVar()
        contentNODEPREP = StringVar()
        contentNESTX = StringVar()
        contentNESTY = StringVar()
        contentSamplingInterval = StringVar()

        content1DEF = StringVar()
        content2DEF = StringVar()
        content3DEF = StringVar()
        content4DEF = StringVar()
        content5DEF = StringVar()
        content6DEF = StringVar()
        content7DEF = StringVar()
        content8DEF = StringVar()
        content9DEF = StringVar()
        content10DEF = StringVar()
        content11DEF = StringVar()
        content12DEF = StringVar()
        content13DEF = StringVar()
        content14DEF = StringVar()
        content15DEF = StringVar()
        content16DEF = StringVar()
        content17DEF = StringVar()
        content18DEF = StringVar()
```

```
content19DEF = StringVar()
content20DEF = StringVar()
content21DEF = StringVar()
content22DEF = StringVar()
content23DEF = StringVar()
content24DEF = StringVar()
content25DEF = StringVar()
content26DEF = StringVar()
content27DEF = StringVar()
contentPPSizesDEF = StringVar()
contentPCoordsDEF = StringVar()
contentNODEPREPDEF = StringVar()
contentNESTXDEF = StringVar()
contentNESTYDEF = StringVar()
contentSamplingIntervalDEF = StringVar()

tempTypesString = ""
tempTypesList = []

# reset individual patch breakdown file
individualPatchBreakdownFile = open('individualPatchBreakdownFile.txt','w')
individualPatchBreakdownFile.close()

# reset precise patch file
precisePatchFile = open('precisePatchFile.txt','w')

precisePatchFile.write("Use this?:")
precisePatchFile.write("\n")
precisePatchFile.write("0")
precisePatchFile.write("\n")

precisePatchFile.close()

# reset patch coordinates file
patchCoordinatesFile = open('patchCoordinatesFile.txt','w')

patchCoordinatesFile.write("Use this?:")
patchCoordinatesFile.write("\n")
patchCoordinatesFile.write("0")
patchCoordinatesFile.write("\n")

patchCoordinatesFile.close()

# read in current parameter values and populate GUI with defaults
batchParametersFile = open('batchParametersFile.txt','r')
batchParametersFile.readline()
content1DEF = batchParametersFile.readline()[:-1]
batchParametersFile.close()

beeParametersFile = open('beeParametersFile.txt','r')
beeParametersFile.readline()
content2DEF = beeParametersFile.readline()[:-1]
beeParametersFile.readline()
contentSamplingIntervalDEF = beeParametersFile.readline()[:-1]
beeParametersFile.readline()
content3DEF = beeParametersFile.readline()[:-1]
beeParametersFile.readline()
beeParametersFile.readline()
beeParametersFile.readline()
beeParametersFile.readline()
beeParametersFile.readline()
content4DEF = beeParametersFile.readline()[:-1]
beeParametersFile.close()

dynamicLandscapeParametersFile = open('dynamicLandscapeParametersFile.txt','r')
dynamicLandscapeParametersFile.readline()
content5DEF = dynamicLandscapeParametersFile.readline()[:-1]
dynamicLandscapeParametersFile.readline()
content6DEF = dynamicLandscapeParametersFile.readline()[:-1]
dynamicLandscapeParametersFile.readline()
content7DEF = dynamicLandscapeParametersFile.readline()[:-1]
dynamicLandscapeParametersFile.readline()
content8DEF = dynamicLandscapeParametersFile.readline()[:-1]
dynamicLandscapeParametersFile.close()

globalParametersFile = open('globalParametersFile.txt','r')
globalParametersFile.readline()
globalParametersFile.readline()
```

```
globalParametersFile.readline()
content9DEF = globalParametersFile.readline()[:-1]
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
content10DEF = globalParametersFile.readline()[:-1]
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
content11DEF = globalParametersFile.readline()[:-1]
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
globalParametersFile.readline()
content12DEF = globalParametersFile.readline()[:-1]
globalParametersFile.readline()
content13DEF = globalParametersFile.readline()[:-1]
globalParametersFile.readline()
contentNODEPREPDEF = globalParametersFile.readline()[:-1]
globalParametersFile.close()

gridParametersFile = open('gridParametersFile.txt','r')
gridParametersFile.readline()
content14DEF = gridParametersFile.readline()[:-1]
gridParametersFile.readline()
content15DEF = gridParametersFile.readline()[:-1]
gridParametersFile.readline()
content16DEF = gridParametersFile.readline()[:-1]
gridParametersFile.readline()
content17DEF = gridParametersFile.readline()[:-1]
gridParametersFile.readline()
content18DEF = gridParametersFile.readline()[:-1]
gridParametersFile.readline()
content19DEF = gridParametersFile.readline()[:-1]
gridParametersFile.close()

modelTypeFile = open('modelTypeFile.txt','r')
modelTypeFile.readline()
content20DEF = modelTypeFile.readline()[:-1]
modelTypeFile.readline()
modelTypeFile.readline()
modelTypeFile.readline()
modelTypeFile.readline()
modelTypeFile.readline()
content21DEF = modelTypeFile.readline()[:-1]
modelTypeFile.close()

typesOfPatchFile = open('typesOfPatchFile.txt','r')
tempTypesString = typesOfPatchFile.readline()[:-1]
tempTypesList = tempTypesString.split(",")
content22DEF = str(tempTypesList[1])
content24DEF = str(tempTypesList[2])
content26DEF = str(tempTypesList[3])
tempTypesString = typesOfPatchFile.readline()[:-1]
tempTypesList = tempTypesString.split(",")
content23DEF = str(tempTypesList[1])
content25DEF = str(tempTypesList[2])
content27DEF = str(tempTypesList[3])
typesOfPatchFile.close()

nestParametersFile = open('nestParametersFile.txt','r')
```

```
        nestParametersFile.readline()
        contentNESTXDEF = nestParametersFile.readline()[:-1]
        nestParametersFile.readline()
        contentNESTYDEF = nestParametersFile.readline()[:-1]
        nestParametersFile.close()

        title1 = Label(self.master, text="BATCH PARAMETERS",fg='blue').grid(row=0, column=0,columnspan=2,sticky=W)
        subTitle1 = Label(self.master, text="Use these parameters to specify the number of trials run for these set of values", fg='blue').grid(row=1, column=0,columnspan=2,sticky=W)

        optionLabel1 = Label(self.master, text="Number of trials to run:").grid(row=2,sticky=W)
        option1 = Entry(self.master,textvariable=content1)
        option1.grid(row=2,column=1,sticky=W)
        option1.insert(0, content1DEF)
        text1 = content1.get()
        content1.set(text1)

        title2 = Label(self.master, text="BEE PARAMETERS",fg='blue').grid(row=3, column=0,columnspan=2,sticky=W)
        subTitle2 = Label(self.master, text="Use these parameters to specify traits of the virtual bee", fg='blue').grid(row=4, column=0,columnspan=2,sticky=W)

        optionLabel2 = Label(self.master, text="Bee Lifespan until Death (foraging bouts):").grid(row=5,sticky=W)
        option2 = Entry(self.master,textvariable=content2)
        option2.grid(row=5,column=1,sticky=W)
        option2.insert(0, content2DEF)
        text2 = content2.get()
        content2.set(text2)
        optionLabel3 = Label(self.master, text="Nectar Carrying Capacity (nectar units):").grid(row=6,sticky=W)
        option3 = Entry(self.master,textvariable=content3)
        option3.grid(row=6,column=1,sticky=W)
        option3.insert(0, content3DEF)
        text3 = content3.get()
        content3.set(text3)
        optionLabel4 = Label(self.master, text="Beta (sensitivity to new experiences; 0.00-1.00):").grid(row=7,sticky=W)
        option4 = Entry(self.master,textvariable=content4)
        option4.grid(row=7,column=1,sticky=W)
        option4.insert(0, content4DEF)
        text4 = content4.get()
        content4.set(text4)

        title3 = Label(self.master, text="LANDSCAPE PARAMETERS",fg='blue').grid(row=8, column=0,columnspan=2,sticky=W)
        subTitle3 = Label(self.master, text="Use these parameters to specify traits of the landscape", fg='blue').grid(row=9, column=0,columnspan=2,sticky=W)

        optionLabel5 = Label(self.master, text="Number of Patches in Landscape:").grid(row=10,sticky=W)
        option5 = Entry(self.master,textvariable=content5)
        option5.grid(row=10,column=1,sticky=W)
        option5.insert(0, content5DEF)
        text5 = content5.get()
        content5.set(text5)

        def confirmPatchNumberChange():
            candidateNumberOfPoorPatches = 0
            candidateNumberOfGoodPatches = 0

            # if number of patches is even
            if (int(content5.get()) % 2) == 0:
                candidateNumberOfPoorPatches = int(content5.get()) / 2
                candidateNumberOfGoodPatches = int(content5.get()) / 2
            else:
                candidateNumberOfPoorPatches = int(content5.get()) / 2
                candidateNumberOfGoodPatches = (int(content5.get()) / 2) + 1

            content24.set(candidateNumberOfPoorPatches)
            content25.set(candidateNumberOfGoodPatches)

        buttonConfirmPatchNumberChange = Button(self.master, text="AUTO-ALLOCATE PATCHES", command=confirmPatchNumberChange, fg='blue')
        buttonConfirmPatchNumberChange.grid(row=9,column=1,sticky=W)

        optionLabel6 = Label(self.master, text="Proposal for Mean Patch Quality at Start:").grid(row=11,sticky=W)
        option6 = Entry(self.master,textvariable=content6)
        option6.grid(row=11,column=1,sticky=W)
        option6.insert(0, content6DEF)
        text6 = content6.get()
        content6.set(text6)
        optionLabel7 = Label(self.master, text="Range in Patch Qualities at Start:").grid(row=12,sticky=W)
        option7 = Entry(self.master,textvariable=content7)
```

```
        option7.grid(row=12,column=1,sticky=W)
        option7.insert(0, content7DEF)
        text7 = content7.get()
        content7.set(text7)
        optionLabel8 = Label(self.master, text="Flight Distance Between Patches (time units):").grid(row=13,sticky=W)
        option8 = Entry(self.master,textvariable=content8)
        option8.grid(row=13,column=1,sticky=W)
        option8.insert(0, content8DEF)
        text8 = content8.get()
        content8.set(text8)

        title4 = Label(self.master, text="GLOBAL PARAMETERS",fg='blue').grid(row=14, column=0,columnspan=2,sticky=W)
        subTitle4 = Label(self.master, text="Use these parameters to specify general traits of the simulation", fg='blue').grid(row=15,
column=0,columnspan=2,sticky=W)

        optionLabel9 = Label(self.master, text="Number of Bees in Simulation:").grid(row=16,sticky=W)
        option9 = Entry(self.master,textvariable=content9)
        option9.grid(row=16,column=1,sticky=W)
        option9.insert(0, content9DEF)
        text9 = content9.get()
        content9.set(text9)
        optionLabel10 = Label(self.master, text="Total Foraging Time (time units):").grid(row=17,sticky=W)
        option10 = Entry(self.master,textvariable=content10)
        option10.grid(row=17,column=1,sticky=W)
        option10.insert(0, content10DEF)
        text10 = content10.get()
        content10.set(text10)
        optionLabel11 = Label(self.master, text="Are Bees OMNISCIENT?:").grid(row=18,sticky=W)
        var11 = IntVar()
        option11 = Checkbutton(self.master, variable=var11)
        option11.grid(row=18,column=1,sticky=W)
        if content11DEF == "1":
            option11.select()
        optionLabel12 = Label(self.master, text="Tick to make nest just 1 time unit from ALL patches:").grid(row=19,sticky=W)
        var12 = IntVar()
        option12 = Checkbutton(self.master, variable=var12)
        option12.grid(row=19,column=1,sticky=W)
        if content12DEF == "1":
            option12.select()
        optionLabel13 = Label(self.master, text="Omega (probability of switch to naivety at nest); 0.00-
1.00):").grid(row=20,sticky=W)
        option13 = Entry(self.master,textvariable=content13)
        option13.grid(row=20,column=1,sticky=W)
        option13.insert(0, content13DEF)
        text13 = content13.get()
        content13.set(text13)
        optionLabelNODEPREP = Label(self.master, text="Tick to turn on nectar depletion and
replenishment:").grid(row=21,sticky=W)
        varNODEPREP = IntVar()
        optionNODEPREP = Checkbutton(self.master, variable=varNODEPREP)
        optionNODEPREP.grid(row=21,column=1,sticky=W)
        if contentNODEPREPDEF == "1":
            optionNODEPREP.select()
        optionLabelNESTX = Label(self.master, text="X-Coordinate of Nest (when non-trivial distance):").grid(row=22,sticky=W)
        varNESTX = IntVar()
        optionNESTX = Entry(self.master,textvariable=contentNESTX)
        optionNESTX.grid(row=22,column=1,sticky=W)
        optionNESTX.insert(0, contentNESTXDEF)
        textNESTX = contentNESTX.get()
        contentNESTX.set(textNESTX)
        optionLabelNESTY = Label(self.master, text="Y-Coordinate of Nest (when non-trivial distance):").grid(row=23,sticky=W)
        varNESTY = IntVar()
        optionNESTY = Entry(self.master,textvariable=contentNESTY)
        optionNESTY.grid(row=23,column=1,sticky=W)
        optionNESTY.insert(0, contentNESTYDEF)
        textNESTY = contentNESTY.get()
        contentNESTY.set(textNESTY)

        title5 = Label(self.master, text="GRID PARAMETERS",fg='blue').grid(row=0,column=2,columnspan=2,sticky=W)
        subTitle5 = Label(self.master, text="Use these parameters to specify the grid of patches and its visualisation",
fg='blue').grid(row=1,column=2,columnspan=2,sticky=W)

        optionLabel14 = Label(self.master, text="Width (in pixels) of the Visualised Grid
Window:").grid(row=2,column=2,sticky=W)
        option14 = Entry(self.master,textvariable=content14)
        option14.grid(row=2,column=3,sticky=W)
        option14.insert(0, content14DEF)
        text14 = content14.get()
```

```
        content14.set(text14)
        optionLabel15 = Label(self.master, text="Height (in pixels) of the Visualised Grid Window:").grid(row=3,column=2,sticky=W)
        option15 = Entry(self.master,textvariable=content15)
        option15.grid(row=3,column=3,sticky=W)
        option15.insert(0, content15DEF)
        text15 = content15.get()
        content15.set(text15)
        optionLabel16 = Label(self.master, text="Size of the Array; Number of Patches Wide:").grid(row=4,column=2,sticky=W)
        option16 = Entry(self.master,textvariable=content16)
        option16.grid(row=4,column=3,sticky=W)
        option16.insert(0, content16DEF)
        text16 = content16.get()
        content16.set(text16)
        optionLabel17 = Label(self.master, text="Size of the Array; Number of Patches High:").grid(row=5,column=2,sticky=W)
        option17 = Entry(self.master,textvariable=content17)
        option17.grid(row=5,column=3,sticky=W)
        option17.insert(0, content17DEF)
        text17 = content17.get()
        content17.set(text17)
        optionLabel18 = Label(self.master, text="Number of seconds between updates on the Visual Grid:").grid(row=6,column=2,sticky=W)
        option18 = Entry(self.master,textvariable=content18)
        option18.grid(row=6,column=3,sticky=W)
        option18.insert(0, content18DEF)
        text18 = content18.get()
        content18.set(text18)
        optionLabel19 = Label(self.master, text="Fading Interval of Tails on Visual Grid:").grid(row=7,column=2,sticky=W)
        option19 = Entry(self.master,textvariable=content19)
        option19.grid(row=7,column=3,sticky=W)
        option19.insert(0, content19DEF)
        text19 = content19.get()
        content19.set(text19)

        title6 = Label(self.master, text="META MODEL PARAMETERS",fg='blue').grid(row=8,column=2,columnspan=2,sticky=W)
        subTitle6 = Label(self.master, text="Use these parameters to specify the type of model to instigate", fg='blue').grid(row=9,column=2,columnspan=2,sticky=W)

        optionLabel20 = Label(self.master, text="Use Cardinal Neighbour Equidistance between patches?:").grid(row=10,column=2,sticky=W)
        var20 = IntVar()
        option20 = Checkbutton(self.master, variable=var20)
        option20.grid(row=10,column=3,sticky=W)
        if content20DEF == "1":
           option20.select()
        optionLabel21 = Label(self.master, text="Automatically Assign Patches to POOR/GOOD Patch Types?:").grid(row=11,column=2,sticky=W)
        var21 = IntVar()
        option21 = Checkbutton(self.master, variable=var21)
        option21.grid(row=11,column=3,sticky=W)
        if content21DEF == "1":
           option21.select()

        title7 = Label(self.master, text="PATCH TYPE PARAMETERS",fg='blue').grid(row=12,column=2,columnspan=2,sticky=W)
        subTitle7 = Label(self.master, text="Use these parameters to specify the types of patch being simulated", fg='blue').grid(row=13,column=2,columnspan=2,sticky=W)
        subTitle7P2 = Label(self.master, text="ONLY alter here if using POOR/GOOD combinations - for anything else, modify config file", fg='blue').grid(row=14,column=2,columnspan=2,sticky=W)

        optionLabel22 = Label(self.master, text="POOR PATCHES:Replenishment Interval (Time Units):").grid(row=15,column=2,sticky=W)
        option22 = Entry(self.master,textvariable=content22)
        option22.grid(row=15,column=3,sticky=W)
        option22.insert(0, content22DEF)
        text22 = content22.get()
        content22.set(text22)

        optionLabel23 = Label(self.master, text="GOOD PATCHES:Replenishment Interval (Time Units):").grid(row=16,column=2,sticky=W)
        option23 = Entry(self.master,textvariable=content23)
        option23.grid(row=16,column=3,sticky=W)
        option23.insert(0, content23DEF)
        text23 = content23.get()
        content23.set(text23)

        optionLabel24 = Label(self.master, text="Total Number of POOR Patches:").grid(row=17,column=2,sticky=W)
```

```
        option24 = Entry(self.master,textvariable=content24)
        option24.grid(row=17,column=3,sticky=W)
        option24.insert(0, content24DEF)
        text24 = content24.get()
        content24.set(text24)

        optionLabel25 = Label(self.master, text="Total Number of GOOD Patches:").grid(row=18,column=2,sticky=W)
        option25 = Entry(self.master,textvariable=content25)
        option25.grid(row=18,column=3,sticky=W)
        option25.insert(0, content25DEF)
        text25 = content25.get()
        content25.set(text25)

        optionLabel26 = Label(self.master, text="Total Flowers Per POOR Patch:").grid(row=19,column=2,sticky=W)
        option26 = Entry(self.master,textvariable=content26)
        option26.grid(row=19,column=3,sticky=W)
        option26.insert(0, content26DEF)
        text26 = content26.get()
        content26.set(text26)

        optionLabel27 = Label(self.master, text="Total Flowers Per GOOD Patch:").grid(row=20,column=2,sticky=W)
        option27 = Entry(self.master,textvariable=content27)
        option27.grid(row=20,column=3,sticky=W)
        option27.insert(0, content27DEF)
        text27 = content27.get()
        content27.set(text27)

        optionLabelSamplingInterval = Label(self.master, text="Flower Handling Time (Time
Units):").grid(row=24,column=0,sticky=W)
        optionSamplingInterval = Entry(self.master,textvariable=contentSamplingInterval)
        optionSamplingInterval.grid(row=24,column=1,sticky=W)
        optionSamplingInterval.insert(0, contentSamplingIntervalDEF)
        textSamplingInterval = contentSamplingInterval.get()
        contentSamplingInterval.set(textSamplingInterval)

        launchStatus.set("Simulation Ready.")

        pleaseWaitLabel = Label(self.master, textvariable=launchStatus, font=("Arial", 16))
        pleaseWaitLabel.grid(row=25,column=0,sticky=W)

        def applyCallback():
            batchParametersFile = open('batchParametersFile.txt','w')
            batchParametersFile.write("Number of Runs:\n")
            batchParametersFile.write(str(content1.get()))
            batchParametersFile.write("\n")
            batchParametersFile.close()

            beeParametersFile = open('beeParametersFile.txt','w')
            beeParametersFile.write("lifespan:\n")
            beeParametersFile.write(str(content2.get()))
            beeParametersFile.write("\n")
            beeParametersFile.write("samplingInterval (must be integer):\n")
            beeParametersFile.write(str(contentSamplingInterval.get()))
            beeParametersFile.write("\n")
            beeParametersFile.write("constantCapacity:\n")
            beeParametersFile.write(str(content3.get()))
            beeParametersFile.write("\n")
            beeParametersFile.write("epsilon:\n")
            beeParametersFile.write("0.0\n")
            beeParametersFile.write("alpha:\n")
            beeParametersFile.write("1.0\n")
            beeParametersFile.write("beta:\n")
            beeParametersFile.write(str(content4.get()))
            beeParametersFile.write("\n")
            beeParametersFile.write("plusRate:\n")
            beeParametersFile.write("1.0\n")
            beeParametersFile.write("minusRate:\n")
            beeParametersFile.write("1.0\n")
            beeParametersFile.close()

            dynamicLandscapeParametersFile = open('dynamicLandscapeParametersFile.txt','w')
            dynamicLandscapeParametersFile.write("Number of Fields in Landscape:\n")
            dynamicLandscapeParametersFile.write(str(content5.get()))
            dynamicLandscapeParametersFile.write("\n")
            dynamicLandscapeParametersFile.write("Proposed Mean Landscape Quality:\n")
            dynamicLandscapeParametersFile.write(str(content6.get()))
            dynamicLandscapeParametersFile.write("\n")
            dynamicLandscapeParametersFile.write("Range in Qualities:\n")
```

```
dynamicLandscapeParametersFile.write(str(content7.get()))
dynamicLandscapeParametersFile.write("\n")
dynamicLandscapeParametersFile.write("Default Landscape Separation:\n")
dynamicLandscapeParametersFile.write(str(content8.get()))
dynamicLandscapeParametersFile.write("\n")
dynamicLandscapeParametersFile.close()

globalParametersFile = open('globalParametersFile.txt','w')
globalParametersFile.write("Bee Injection Interval:\n")
globalParametersFile.write("0\n")
globalParametersFile.write("Maximum Injections:\n")
globalParametersFile.write(str(content9.get()))
globalParametersFile.write("\n")
globalParametersFile.write("Time to Complete AT NEST:\n")
globalParametersFile.write("0\n")
globalParametersFile.write("Simulation Cut-Off:\n")
globalParametersFile.write(str(content10.get()))
globalParametersFile.write("\n")
globalParametersFile.write("Action Value Tracking On / Off:\n")
globalParametersFile.write("0\n")
globalParametersFile.write("Action Value / Est Qual / Location / Rem Cap Tracking Bee :\n")
globalParametersFile.write("0\n")
globalParametersFile.write("DEPLETION Constant:\n")
globalParametersFile.write("1\n")
globalParametersFile.write("Mean Uptake Over Time Calculation On / Off:\n")
globalParametersFile.write("0\n")
globalParametersFile.write("Sudden Resource Injection On / Off:\n")
globalParametersFile.write("0\n")
globalParametersFile.write("Sudden Resource Injection Amount:\n")
globalParametersFile.write("5000\n")
globalParametersFile.write("Sudden Resource Injection Time:\n")
globalParametersFile.write("30000\n")
globalParametersFile.write("Sudden Resource Injection Field (A = 0, B = 1, C = 2)\n")
globalParametersFile.write("1\n")
globalParametersFile.write("Colony Omniscience (0 = OFF, 1 = ON):\n")
globalParametersFile.write(str(var11.get()))
globalParametersFile.write("\n")
globalParametersFile.write("PAUSE to Check Parameters:\n")
globalParametersFile.write("0\n")
globalParametersFile.write("Minimal Log:\n")
globalParametersFile.write("1\n")
globalParametersFile.write("Gamma:\n")
globalParametersFile.write("0.01\n")
globalParametersFile.write("Trivial-Distance Nest (0 = OFF, 1 = ON)\n")
globalParametersFile.write(str(var12.get()))
globalParametersFile.write("\n")
globalParametersFile.write("Omega:\n")
globalParametersFile.write(str(content13.get()))
globalParametersFile.write("\n")
globalParametersFile.write("Use Nectar Depletion and Replenishment:\n")
globalParametersFile.write(str(varNODEPREP.get()))
globalParametersFile.write("\n")

globalParametersFile.close()

gridParametersFile = open('gridParametersFile.txt','w')
gridParametersFile.write("Canvas Width\n")
gridParametersFile.write(str(content14.get()))
gridParametersFile.write("\n")
gridParametersFile.write("Canvas Height\n")
gridParametersFile.write(str(content15.get()))
gridParametersFile.write("\n")
gridParametersFile.write("Grid Width (Cells)\n")
gridParametersFile.write(str(content16.get()))
gridParametersFile.write("\n")
gridParametersFile.write("Grid Height (Cells)\n")
gridParametersFile.write(str(content17.get()))
gridParametersFile.write("\n")
gridParametersFile.write("Visual Delay (real time time-step representation on visual grid)\n")
gridParametersFile.write(str(content18.get()))
gridParametersFile.write("\n")
gridParametersFile.write("Fading Interval (in visual update cycles)\n")
gridParametersFile.write(str(content19.get()))
gridParametersFile.write("\n")
gridParametersFile.close()

modelTypeFile = open('modelTypeFile.txt','w')
```

```
            modelTypeFile.write("Distance Calculation (0 = GLOBAL EQUIDISTANCE, 1 = NEIGHBORHOOD EQUIDISTANCE):\n")
            modelTypeFile.write(str(var20.get()))
            modelTypeFile.write("\n")
            modelTypeFile.write("Estimated Replenishment Model (0 = EXPONENTIAL MODEL, 1 = LINEAR MODEL, any other number = no estimated replenishment model)\n")
            modelTypeFile.write("2\n")
            modelTypeFile.write("Cap Estimated Quality at 1.0? (0 = no, 1 = YES)\n")
            modelTypeFile.write("0\n")
            modelTypeFile.write("Auto-assign Patch Types? [only use for poor/good combination : EVENS = poor, ODDS = good] (0 = no, 1 = yes)\n")
            modelTypeFile.write(str(var21.get()))
            modelTypeFile.write("\n")
            modelTypeFile.close()

            typesOfPatchFile = open('typesOfPatchFile.txt','w')
            typesOfPatchFile.write("Poor,")
            typesOfPatchFile.write(str(content22.get()))
            typesOfPatchFile.write(",")
            typesOfPatchFile.write(str(content24.get()))
            typesOfPatchFile.write(",")
            typesOfPatchFile.write(str(content26.get()))
            typesOfPatchFile.write("\n")
            typesOfPatchFile.write("Good,")
            typesOfPatchFile.write(str(content23.get()))
            typesOfPatchFile.write(",")
            typesOfPatchFile.write(str(content25.get()))
            typesOfPatchFile.write(",")
            typesOfPatchFile.write(str(content27.get()))
            typesOfPatchFile.write("\n")
            typesOfPatchFile.close()

            nestParametersFile = open('nestParametersFile.txt','w')
            nestParametersFile.write("X-Coordinate of Nest: (only used when trivial distance nest switched off)")
            nestParametersFile.write("\n")
            nestParametersFile.write(str(contentNESTX.get()))
            nestParametersFile.write("\n")
            nestParametersFile.write("Y-Coordinate of Nest: (only used when trivial distance nest switched off)")
            nestParametersFile.write("\n")
            nestParametersFile.write(str(contentNESTY.get()))
            nestParametersFile.write("\n")
            nestParametersFile.close()

        def launchHARVEST():
            applyCallback()

            launchStatus.set("Simulation Running - Please Wait")
            pleaseWaitLabel.configure(fg='red')
            buttonApply.configure(state=DISABLED)
            buttonlaunchHARVEST.configure(state=DISABLED)
            buttonSpecifyPrecisePatchSizes.configure(state=DISABLED)
            buttonSpecifyPatchCoordinates.configure(state=DISABLED)

            HARVESTThread().start()

        def specifyPrecisePatchSizes():
            # create scrolled canvas

            top3 = Toplevel()

            vscrollbar = AutoScrollbar(top3)
            vscrollbar.grid(row=0, column=1, sticky=N+S)
            hscrollbar = AutoScrollbar(top3, orient=HORIZONTAL)
            hscrollbar.grid(row=1, column=0, sticky=E+W)

            canvasPatchSizes = Canvas(top3,yscrollcommand=vscrollbar.set,xscrollcommand=hscrollbar.set)
            canvasPatchSizes.grid(row=0, column=0, sticky=N+S+E+W)

            vscrollbar.config(command=canvasPatchSizes.yview)
            hscrollbar.config(command=canvasPatchSizes.xview)

            # make the canvas expandable
            top3.grid_rowconfigure(0, weight=1)
            top3.grid_columnconfigure(0, weight=1)

            # create canvas contents

            frame1 = Frame(canvasPatchSizes)
```

```
    frame1.rowconfigure(1, weight=1)
    frame1.columnconfigure(1, weight=1)

    listOfTextVarsPPFULLs = []
    listOfTextVarsPPTOTALs = []

    rows1 = int(content5.get()) + 2

    Label(frame1, text="Tick to use these settings instead of basic selection in main window:").grid(row=1,column=0)
    varPPSizes = IntVar()
    optionPPSizes = Checkbutton(frame1, variable=varPPSizes).grid(row=1,column=1)
    if contentPPSizesDEF == "1":
       optionPPSizes.select()

    for i in range(2,rows1):
       Label(frame1, text=("Patch " + str((i - 1)) + " FULLs 1st box, TOTAL 2nd box :")).grid(row=i,column=0)
       listOfTextVarsPPFULLs.append(StringVar())
       Entry(frame1,textvariable=listOfTextVarsPPFULLs[(len(listOfTextVarsPPFULLs) - 1)]).grid(row=i,column=1)
       listOfTextVarsPPTOTALs.append(StringVar())
       Entry(frame1,textvariable=listOfTextVarsPPTOTALs[(len(listOfTextVarsPPTOTALs) - 1)]).grid(row=i,column=2)

    canvasPatchSizes.create_window(0, 0, anchor=NW, window=frame1)

    frame1.update_idletasks()

    canvasPatchSizes.config(scrollregion=canvasPatchSizes.bbox("all"))

    def savePrecisePatchChanges():
       precisePatchFile = open('precisePatchFile.txt','w')
       precisePatchFile.write("Use this?:")
       precisePatchFile.write("\n")
       precisePatchFile.write(str(varPPSizes.get()))
       precisePatchFile.write("\n")

       for i in range(1,(rows1 - 1)):
          precisePatchFile.write("Patch ")
          precisePatchFile.write(str(i))
          precisePatchFile.write("\n")
          precisePatchFile.write(str(listOfTextVarsPPFULLs[(i-1)].get()))
          precisePatchFile.write("\n")
          precisePatchFile.write(str(listOfTextVarsPPTOTALs[(i-1)].get()))
          precisePatchFile.write("\n")

       precisePatchFile.close()

    Button(frame1, text="Save Changes", command=savePrecisePatchChanges).grid(row=rows1,column=0)

def specifyPatchCoordinates():
    # create scrolled canvas

    top4 = Toplevel()

    vscrollbar2 = AutoScrollbar(top4)
    vscrollbar2.grid(row=0, column=1, sticky=N+S)
    hscrollbar2 = AutoScrollbar(top4, orient=HORIZONTAL)
    hscrollbar2.grid(row=1, column=0, sticky=E+W)

    canvasPatchCoords = Canvas(top4,yscrollcommand=vscrollbar2.set,xscrollcommand=hscrollbar2.set)
    canvasPatchCoords.grid(row=0, column=0, sticky=N+S+E+W)

    vscrollbar2.config(command=canvasPatchCoords.yview)
    hscrollbar2.config(command=canvasPatchCoords.xview)

    # make the canvas expandable
    top4.grid_rowconfigure(0, weight=1)
    top4.grid_columnconfigure(0, weight=1)

    # create canvas contents

    frame2 = Frame(canvasPatchCoords)
    frame2.rowconfigure(1, weight=1)
    frame2.columnconfigure(1, weight=1)

    listOfTextVarsPCOORDXs = []
    listOfTextVarsPCOORDYs = []

    rows2 = int(content5.get()) + 2
```

```
            Label(frame2, text="Tick to use these settings instead of basic selection in main window:").grid(row=1,column=0)
            varPCoords = IntVar()
            optionPCoords = Checkbutton(frame2, variable=varPCoords).grid(row=1,column=1)
            if contentPCoords == "1":
               optionPCoords.select()

            for i in range(2,rows2):
               Label(frame2, text=("Patch " + str((i - 1)) + " X-Coordinate 1st box, Y-Coordinate 2nd box :")).grid(row=i,column=0)
               listOfTextVarsPCOORDXs.append(StringVar())
               Entry(frame2,textvariable=listOfTextVarsPCOORDXs[(len(listOfTextVarsPCOORDXs) - 1)]).grid(row=i,column=1)
               listOfTextVarsPCOORDYs.append(StringVar())
               Entry(frame2,textvariable=listOfTextVarsPCOORDYs[(len(listOfTextVarsPCOORDYs) - 1)]).grid(row=i,column=2)

            canvasPatchCoords.create_window(0, 0, anchor=NW, window=frame2)

            frame2.update_idletasks()

            canvasPatchCoords.config(scrollregion=canvasPatchCoords.bbox("all"))

            def saveCoordinateChanges():
               patchCoordinatesFile = open('patchCoordinatesFile.txt','w')
               patchCoordinatesFile.write("Use this?:")
               patchCoordinatesFile.write("\n")
               patchCoordinatesFile.write(str(varPCoords.get()))
               patchCoordinatesFile.write("\n")

               for i in range(1,(rows2 - 1)):
                  patchCoordinatesFile.write("Patch ")
                  patchCoordinatesFile.write(str(i))
                  patchCoordinatesFile.write("\n")
                  patchCoordinatesFile.write(str(listOfTextVarsPCOORDXs[(i-1)].get()))
                  patchCoordinatesFile.write(",")
                  patchCoordinatesFile.write(str(listOfTextVarsPCOORDYs[(i-1)].get()))
                  patchCoordinatesFile.write("\n")

               patchCoordinatesFile.close()

            Button(frame2, text="Save Changes", command=saveCoordinateChanges).grid(row=rows2,column=0)
            Label(frame2, text="***NOTE - If nest equidistance from all patches NOT used then nest assumed to be at (0,0) if these
overrides are used",fg='blue').grid(row=rows2+1,column=0)

         buttonSpecifyPrecisePatchSizes = Button(self.master, text="Specify Precise Patch Sizes",
command=specifyPrecisePatchSizes)
         buttonSpecifyPrecisePatchSizes.grid(row=21,column=3,sticky=W)
         buttonSpecifyPatchCoordinates = Button(self.master, text="Specify Patch Coordinates", command=specifyPatchCoordinates)
         buttonSpecifyPatchCoordinates.grid(row=22,column=3,sticky=W)
         buttonApply = Button(self.master, text="Apply",command=applyCallback)
         buttonApply.grid(row=23,column=3,sticky=W)
         buttonlaunchHARVEST = Button(self.master, text="Launch HARVEST",command=launchHARVEST)
         buttonlaunchHARVEST.grid(row=24,column=3,sticky=W)

class HARVESTThread(threading.Thread):
   def run(self):
      import HARVEST
      HARVEST.startProgram()

      calculateSE = 1

      se1Value = 0.0
      se2Value = 0.0
      se3Value = 0.0
      se4Value = 0.0
      se5Value = 0.0
      se6Value = 0.0
      se7Value = 0.0
      se8Value = 0.0
      se9Value = 0.0
      se10Value = 0.0
      se11Value = 0.0

      # check if only single trial being run so can decide whether to calculate standard errors
      batchParametersFile = open('batchParametersFile.txt','r')

      batchParametersFile.readline()
      tempTrialsThisTime = int(batchParametersFile.readline())

      if tempTrialsThisTime == 1:
         calculateSE = 0
```

```
batchParametersFile.close()

# calculate mean traffic per foraging bout
meanTrafficFile = open('meanTrafficFile.txt','r')

tempRead = meanTrafficFile.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result1Value = 0.0

for i in range(0, len(tempList)):
   tempSum = float(tempSum) + float(tempList[i])

result1Value = float(tempSum) / float(len(tempList))

meanTrafficFile.close()

if calculateSE == 1:
   # calculate standard error
   tempSqDiff = 0.0
   tempSum = 0.0
   se1Value = 0.0

   for i in range(0, len(tempList)):
      tempSqDiff = float(tempList[i]) - float(result1Value)
      tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
      tempSum = float(tempSum) + float(tempSqDiff)

   tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
   tempSum = float(math.sqrt(tempSum))
   se1Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate mean total traffic
meanTOTALTrafficFile = open('meanTOTALTrafficFile.txt','r')

tempRead = meanTOTALTrafficFile.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result2Value = 0.0

for i in range(0, len(tempList)):
   tempSum = float(tempSum) + float(tempList[i])

result2Value = float(tempSum) / float(len(tempList))

meanTOTALTrafficFile.close()

if calculateSE == 1:
   # calculate standard error
   tempSqDiff = 0.0
   tempSum = 0.0
   se2Value = 0.0

   for i in range(0, len(tempList)):
      tempSqDiff = float(tempList[i]) - float(result2Value)
      tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
      tempSum = float(tempSum) + float(tempSqDiff)

   tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
   tempSum = float(math.sqrt(tempSum))
   se2Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate mean unique patches visited
meanUniquePatchesVisited = open('meanUniquePatchesVisited.txt','r')

tempRead = meanUniquePatchesVisited.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result3Value = 0.0

for i in range(0, len(tempList)):
   tempSum = float(tempSum) + float(tempList[i])

result3Value = float(tempSum) / float(len(tempList))

meanUniquePatchesVisited.close()
```

```
if calculateSE == 1:
   # calculate standard error
   tempSqDiff = 0.0
   tempSum = 0.0
   se3Value = 0.0

   for i in range(0, len(tempList)):
      tempSqDiff = float(tempList[i]) - float(result3Value)
      tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
      tempSum = float(tempSum) + float(tempSqDiff)

   tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
   tempSum = float(math.sqrt(tempSum))
   se3Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate mean bouts completed
meanBoutsCompletedFile = open('meanBoutsCompletedFile.txt','r')

tempRead = meanBoutsCompletedFile.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result4Value = 0.0

for i in range(0, len(tempList)):
   tempSum = float(tempSum) + float(tempList[i])

result4Value = float(tempSum) / float(len(tempList))

meanBoutsCompletedFile.close()

if calculateSE == 1:
   # calculate standard error
   tempSqDiff = 0.0
   tempSum = 0.0
   se4Value = 0.0

   for i in range(0, len(tempList)):
      tempSqDiff = float(tempList[i]) - float(result4Value)
      tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
      tempSum = float(tempSum) + float(tempSqDiff)

   tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
   tempSum = float(math.sqrt(tempSum))
   se4Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate mean foraging performance
meanPerformanceFile = open('meanPerformanceFile.txt','r')

tempRead = meanPerformanceFile.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result5Value = 0.0

for i in range(0, len(tempList)):
   tempSum = float(tempSum) + float(tempList[i])

result5Value = float(tempSum) / float(len(tempList))

meanPerformanceFile.close()

if calculateSE == 1:
   # calculate standard error
   tempSqDiff = 0.0
   tempSum = 0.0
   se5Value = 0.0

   for i in range(0, len(tempList)):
      tempSqDiff = float(tempList[i]) - float(result5Value)
      tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
      tempSum = float(tempSum) + float(tempSqDiff)

   tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
   tempSum = float(math.sqrt(tempSum))
   se5Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate residence (ALL)
meanResidenceFileALL = open('meanResidenceFileALL.txt','r')
```

```
tempRead = meanResidenceFileALL.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result6Value = 0.0

for i in range(0, len(tempList)):
   tempSum = float(tempSum) + float(tempList[i])

result6Value = float(tempSum) / float(len(tempList))

meanResidenceFileALL.close()

if calculateSE == 1:
   # calculate standard error
   tempSqDiff = 0.0
   tempSum = 0.0
   se6Value = 0.0

   for i in range(0, len(tempList)):
      tempSqDiff = float(tempList[i]) - float(result6Value)
      tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
      tempSum = float(tempSum) + float(tempSqDiff)

   tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
   tempSum = float(math.sqrt(tempSum))
   se6Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate residence (POOR)
meanResidenceFilePOOR = open('meanResidenceFilePOOR.txt','r')

tempRead = meanResidenceFilePOOR.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result7Value = 0.0

for i in range(0, len(tempList)):
   tempSum = float(tempSum) + float(tempList[i])

result7Value = float(tempSum) / float(len(tempList))

meanResidenceFilePOOR.close()

if calculateSE == 1:
   # calculate standard error
   tempSqDiff = 0.0
   tempSum = 0.0
   se7Value = 0.0

   for i in range(0, len(tempList)):
      tempSqDiff = float(tempList[i]) - float(result7Value)
      tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
      tempSum = float(tempSum) + float(tempSqDiff)

   tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
   tempSum = float(math.sqrt(tempSum))
   se7Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate residence (GOOD)
meanResidenceFileGOOD = open('meanResidenceFileGOOD.txt','r')

tempRead = meanResidenceFileGOOD.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result8Value = 0.0

for i in range(0, len(tempList)):
   tempSum = float(tempSum) + float(tempList[i])

result8Value = float(tempSum) / float(len(tempList))

meanResidenceFileGOOD.close()

if calculateSE == 1:
   # calculate standard error
   tempSqDiff = 0.0
   tempSum = 0.0
   se8Value = 0.0
```

```
    for i in range(0, len(tempList)):
        tempSqDiff = float(tempList[i]) - float(result8Value)
        tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
        tempSum = float(tempSum) + float(tempSqDiff)

    tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
    tempSum = float(math.sqrt(tempSum))
    se8Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate visits (ALL)
meanALLVisitsFile = open('meanALLVisitsFile.txt','r')

tempRead = meanALLVisitsFile.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result9Value = 0.0

for i in range(0, len(tempList)):
    tempSum = float(tempSum) + float(tempList[i])

result9Value = float(tempSum) / float(len(tempList))

meanALLVisitsFile.close()

if calculateSE == 1:
    # calculate standard error
    tempSqDiff = 0.0
    tempSum = 0.0
    se9Value = 0.0

    for i in range(0, len(tempList)):
        tempSqDiff = float(tempList[i]) - float(result9Value)
        tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
        tempSum = float(tempSum) + float(tempSqDiff)

    tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
    tempSum = float(math.sqrt(tempSum))
    se9Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate visits (POOR)
meanPOORVisitsFile = open('meanPOORVisitsFile.txt','r')

tempRead = meanPOORVisitsFile.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result10Value = 0.0

for i in range(0, len(tempList)):
    tempSum = float(tempSum) + float(tempList[i])

result10Value = float(tempSum) / float(len(tempList))

meanPOORVisitsFile.close()

if calculateSE == 1:
    # calculate standard error
    tempSqDiff = 0.0
    tempSum = 0.0
    se10Value = 0.0

    for i in range(0, len(tempList)):
        tempSqDiff = float(tempList[i]) - float(result10Value)
        tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
        tempSum = float(tempSum) + float(tempSqDiff)

    tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
    tempSum = float(math.sqrt(tempSum))
    se10Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

# calculate visits (GOOD)
meanGOODVisitsFile = open('meanGOODVisitsFile.txt','r')

tempRead = meanGOODVisitsFile.read()
tempList = tempRead.splitlines()
tempSum = 0.0
result11Value = 0.0

for i in range(0, len(tempList)):
```

```
    tempSum = float(tempSum) + float(tempList[i])

result11Value = float(tempSum) / float(len(tempList))

meanGOODVisitsFile.close()

if calculateSE == 1:
    # calculate standard error
    tempSqDiff = 0.0
    tempSum = 0.0
    se11Value = 0.0

    for i in range(0, len(tempList)):
        tempSqDiff = float(tempList[i]) - float(result11Value)
        tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
        tempSum = float(tempSum) + float(tempSqDiff)

    tempSum = float(tempSum) / float((float(len(tempList)) - 1.0))
    tempSum = float(math.sqrt(tempSum))
    se11Value = float(tempSum) / float(math.sqrt(float(len(tempList))))

top = Toplevel()

Otitle1 = Label(top, text="Traffic Results",fg='red',font=("Arial", 16))
Otitle1.grid(row=0,column=0,columnspan=2,sticky=W)

resultLabel1 = Label(top, text="Mean Traffic Per Bee Per Foraging Bout:",fg='blue')
resultLabel1.grid(row=1,column=0,sticky=W)
result1 = Label(top, text=str(round(result1Value,3)))
result1.grid(row=1,column=1,sticky=W)
seLabel1 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel1.grid(row=1,column=2,sticky=W)
se1 = Label(top, text=str(round(se1Value, 3)))
se1.grid(row=1,column=3,sticky=W)

resultLabel2 = Label(top, text="Mean Traffic Per Bee In Trial:",fg='blue')
resultLabel2.grid(row=2,column=0,sticky=W)
result2 = Label(top, text=str(round(result2Value,3)))
result2.grid(row=2,column=1,sticky=W)
seLabel2 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel2.grid(row=2,column=2,sticky=W)
se2 = Label(top, text=str(round(se2Value, 3)))
se2.grid(row=2,column=3,sticky=W)

resultLabel3 = Label(top, text="Mean Number of Unique Patches Visited Per Bee:",fg='blue')
resultLabel3.grid(row=3,column=0,sticky=W)
result3 = Label(top, text=str(round(result3Value,3)))
result3.grid(row=3,column=1,sticky=W)
seLabel3 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel3.grid(row=3,column=2,sticky=W)
se3 = Label(top, text=str(round(se3Value, 3)))
se3.grid(row=3,column=3,sticky=W)

resultLabel4 = Label(top, text="Mean Number of Foraging Bouts Completed Per Bee:",fg='blue')
resultLabel4.grid(row=4,column=0,sticky=W)
result4 = Label(top, text=str(round(result4Value,3)))
result4.grid(row=4,column=1,sticky=W)
seLabel4 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel4.grid(row=4,column=2,sticky=W)
se4 = Label(top, text=(round(se4Value, 3)))
se4.grid(row=4,column=3,sticky=W)

Otitle2 = Label(top, text="Performance Results",fg='red',font=("Arial", 16))
Otitle2.grid(row=5,column=0,columnspan=2,sticky=W)

resultLabel5 = Label(top, text="Mean Foraging Performance (Rewards/Time Taken) Per Bee:",fg='blue')
resultLabel5.grid(row=6,column=0,sticky=W)
result5 = Label(top, text=str(round(result5Value,3)))
result5.grid(row=6,column=1,sticky=W)
seLabel5 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel5.grid(row=6,column=2,sticky=W)
se5 = Label(top, text=(round(se5Value, 3)))
se5.grid(row=6,column=3,sticky=W)

Otitle3 = Label(top, text="Residence Results",fg='red',font=("Arial", 16))
Otitle3.grid(row=7,column=0,columnspan=2,sticky=W)

resultLabel6 = Label(top, text="Mean Patch Residence (no. of flowers) Per Bee:",fg='blue')
```

```
resultLabel6.grid(row=8,column=0,sticky=W)
result6 = Label(top, text=str(round(result6Value,3)))
result6.grid(row=8,column=1,sticky=W)
seLabel6 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel6.grid(row=8,column=2,sticky=W)
se6 = Label(top, text=(round(se6Value, 3)))
se6.grid(row=8,column=3,sticky=W)

resultLabel7 = Label(top, text="Mean Patch Residence IN POOR PATCHES Per Bee:",fg='blue')
resultLabel7.grid(row=9,column=0,sticky=W)
result7 = Label(top, text=str(round(result7Value,3)))
result7.grid(row=9,column=1,sticky=W)
seLabel7 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel7.grid(row=9,column=2,sticky=W)
se7 = Label(top, text=(round(se7Value, 3)))
se7.grid(row=9,column=3,sticky=W)

resultLabel8 = Label(top, text="Mean Patch Residence IN GOOD PATCHES Per Bee:",fg='blue')
resultLabel8.grid(row=10,column=0,sticky=W)
result8 = Label(top, text=(round(result8Value,3)))
result8.grid(row=10,column=1,sticky=W)
seLabel8 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel8.grid(row=10,column=2,sticky=W)
se8 = Label(top, text=(round(se8Value, 3)))
se8.grid(row=10,column=3,sticky=W)

resultLabel9 = Label(top, text="Mean Total Patch Visits Per Bee:",fg='blue')
resultLabel9.grid(row=11,column=0,sticky=W)
result9 = Label(top, text=(round(result9Value,3)))
result9.grid(row=11,column=1,sticky=W)
seLabel9 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel9.grid(row=11,column=2,sticky=W)
se9 = Label(top, text=(round(se9Value, 3)))
se9.grid(row=11,column=3,sticky=W)

resultLabel10 = Label(top, text="Mean POOR Patch Visits Per Bee:",fg='blue')
resultLabel10.grid(row=12,column=0,sticky=W)
result10 = Label(top, text=(round(result10Value,3)))
result10.grid(row=12,column=1,sticky=W)
seLabel10 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel10.grid(row=12,column=2,sticky=W)
se10 = Label(top, text=(round(se10Value, 3)))
se10.grid(row=12,column=3,sticky=W)

resultLabel11 = Label(top, text="Mean GOOD Patch Visits Per Bee:",fg='blue')
resultLabel11.grid(row=13,column=0,sticky=W)
result11 = Label(top, text=(round(result11Value,3)))
result11.grid(row=13,column=1,sticky=W)
seLabel11 = Label(top, text="S.E. (Across Trials):",fg='blue')
seLabel11.grid(row=13,column=2,sticky=W)
se11 = Label(top, text=(round(se11Value, 3)))
se11.grid(row=13,column=3,sticky=W)

def launchWatchBeeMovements():
    HARVEST.startGridPlayer()

def launchViewTransitionMatrix():
    top2 = Toplevel()

    tempTransitionMatrixString = ""
    transitionMatrixFile = open('transitionMatrixFile.txt','r')
    tempTransitionMatrixString = transitionMatrixFile.read()
    transitionMatrixFile.close()

    scrollbar = Scrollbar(top2)
    scrollbar.pack(side=RIGHT, fill=Y)

    scrollbar2 = Scrollbar(top2, orient=HORIZONTAL)
    scrollbar2.pack(side=BOTTOM, fill=X)

    transitionMatrixText = Text(top2, wrap=NONE, xscrollcommand=scrollbar2.set, yscrollcommand=scrollbar.set)
    transitionMatrixText.pack()
    transitionMatrixText.insert(END, tempTransitionMatrixString)

    scrollbar2.config(command=transitionMatrixText.xview)
    scrollbar.config(command=transitionMatrixText.yview)

def launchViewIndividualPatchBreakdown():
```

```
top5 = Toplevel()

vscrollbar3 = AutoScrollbar(top5)
vscrollbar3.grid(row=0, column=1, sticky=N+S)
hscrollbar3 = AutoScrollbar(top5, orient=HORIZONTAL)
hscrollbar3.grid(row=1, column=0, sticky=E+W)

canvasPatchBreakdown = Canvas(top5,yscrollcommand=vscrollbar3.set,xscrollcommand=hscrollbar3.set)
canvasPatchBreakdown.grid(row=0, column=0, sticky=N+S+E+W)

vscrollbar3.config(command=canvasPatchBreakdown.yview)
hscrollbar3.config(command=canvasPatchBreakdown.xview)

# make the canvas expandable
top5.grid_rowconfigure(0, weight=1)
top5.grid_columnconfigure(0, weight=1)

# create canvas contents

frame3 = Frame(canvasPatchBreakdown)
frame3.rowconfigure(1, weight=1)
frame3.columnconfigure(1, weight=1)

# read in individual patch residences
meanResidenceFileEACH = open('meanResidenceFileEACH.txt','r')

tempEACHRESString = meanResidenceFileEACH.read()

meanResidenceFileEACH.close()

tempEACHRESList1 = tempEACHRESString.splitlines()
tempEACHRESList2 = []

for i in range(0, len(tempEACHRESList1)):
   tempEACHRESList2.append(tempEACHRESList1[i].split(","))
   tempEACHRESList2[i].pop()          # remove last entry which is blank

# read in individual patch visits
meanEACHVisitsFile = open('meanEACHVisitsFile.txt','r')

tempEACHVISITSString = meanEACHVisitsFile.read()

meanEACHVisitsFile.close()

tempEACHVISITSList1 = tempEACHVISITSString.splitlines()
tempEACHVISITSList2 = []

for i in range(0, len(tempEACHVISITSList1)):
   tempEACHVISITSList2.append(tempEACHVISITSList1[i].split(","))
   tempEACHVISITSList2[i].pop()          # remove last entry which is blank

# work out mean res across trials
meanResEACH = []
meanResEACHRunning = 0.0

for i in range(0, len(tempEACHRESList2[0])):
   meanResEACHRunning = 0.0

   for j in range(0, len(tempEACHRESList2)):
      meanResEACHRunning = float(meanResEACHRunning) + float(tempEACHRESList2[j][i])

   meanResEACHRunning = float(meanResEACHRunning) / float(len(tempEACHRESList2))

   meanResEACH.append(float(meanResEACHRunning))

# work out mean visits across trials
meanVisitsEACH = []
meanVisitsEACHRunning = 0.0

for i in range(0, len(tempEACHVISITSList2[0])):
   meanVisitsEACHRunning = 0.0

   for j in range(0, len(tempEACHVISITSList2)):
      meanVisitsEACHRunning = float(meanVisitsEACHRunning) + float(tempEACHVISITSList2[j][i])

   meanVisitsEACHRunning = float(meanVisitsEACHRunning) / float(len(tempEACHVISITSList2))

   meanVisitsEACH.append(float(meanVisitsEACHRunning))
```

```
            # work out standard errors
            SEResidence = []
            SEVisits = []

            if calculateSE == 1:
               for i in range(0, len(tempEACHRESList2[0])):
                  tempSqDiff = 0.0
                  tempSum = 0.0
                  seValue = 0.0
                  for j in range(0, len(tempEACHRESList2)):
                     tempSqDiff = float(tempEACHRESList2[j][i]) - float(meanResEACH[i])
                     tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
                     tempSum = float(tempSum) + float(tempSqDiff)

                  tempSum = float(tempSum) / float((float(len(tempEACHRESList2)) - 1.0))
                  tempSum = float(math.sqrt(tempSum))
                  seValue = float(tempSum) / float(math.sqrt(float(len(tempEACHRESList2))))

                  SEResidence.append(seValue)

               for i in range(0, len(tempEACHVISITSList2[0])):
                  tempSqDiff = 0.0
                  tempSum = 0.0
                  seValue = 0.0
                  for j in range(0, len(tempEACHVISITSList2)):
                     tempSqDiff = float(tempEACHVISITSList2[j][i]) - float(meanVisitsEACH[i])
                     tempSqDiff = float(tempSqDiff) * float(tempSqDiff)
                     tempSum = float(tempSum) + float(tempSqDiff)

                  tempSum = float(tempSum) / float((float(len(tempEACHVISITSList2)) - 1.0))
                  tempSum = float(math.sqrt(tempSum))
                  seValue = float(tempSum) / float(math.sqrt(float(len(tempEACHVISITSList2))))

                  SEVisits.append(seValue)
            else:
               for i in range(0, len(tempEACHRESList2[0])):
                  SEResidence.append(0.0)

               for i in range(0, len(tempEACHVISITSList2[0])):
                  SEVisits.append(0.0)

            # set up GUI
            rows3 = len(tempEACHRESList2[0]) + 1

            Label(frame3,text="Means over trials, S.E. shown in brackets").grid(row=0)

            individualPatchBreakdownFile = open('individualPatchBreakdownFile.txt','w')
            individualPatchBreakdownFile.write("FORMAT = Patch No., Mean Residence Per Bee, S.E, Mean Visits Per Bee, S.E")
            individualPatchBreakdownFile.write("\n")

            for i in range(1,rows3):
               Label(frame3, text=("Patch " + str(i) + ": MEAN RESIDENCE = " + str(round((meanResEACH[(i-1)]),3)) + " (" +
str(round((SEResidence[(i-1)]),3)) + ") " + "  MEAN VISITS = " + str(round((meanVisitsEACH[(i-1)]),3)) + " (" +
str(round((SEVisits[(i-1)]),3)) + ") ")).grid(row=i)
               individualPatchBreakdownFile.write(str(i))
               individualPatchBreakdownFile.write(",")
               individualPatchBreakdownFile.write(str(round((meanResEACH[(i-1)]),3)))
               individualPatchBreakdownFile.write(",")
               individualPatchBreakdownFile.write(str(round((SEResidence[(i-1)]),3)))
               individualPatchBreakdownFile.write(",")
               individualPatchBreakdownFile.write(str(round((meanVisitsEACH[(i-1)]),3)))
               individualPatchBreakdownFile.write(",")
               individualPatchBreakdownFile.write(str(round((SEVisits[(i-1)]),3)))
               individualPatchBreakdownFile.write("\n")

            individualPatchBreakdownFile.close()

            canvasPatchBreakdown.create_window(0, 0, anchor=NW, window=frame3)

            frame3.update_idletasks()

            canvasPatchBreakdown.config(scrollregion=canvasPatchBreakdown.bbox("all"))

         def launchViewNaivetyReport():
            top6 = Toplevel()

            tempNaivetyReportString = ""
```

```
        naivetyReport = open('naivetyReport.txt','r')
        tempNaivetyReportString = naivetyReport.read()
        naivetyReport.close()

        scrollbarNaive = Scrollbar(top6)
        scrollbarNaive.pack(side=RIGHT, fill=Y)

        scrollbar2Naive = Scrollbar(top6, orient=HORIZONTAL)
        scrollbar2Naive.pack(side=BOTTOM, fill=X)

        naivetyReportText = Text(top6, wrap=NONE, xscrollcommand=scrollbar2Naive.set, yscrollcommand=scrollbarNaive.set)
        naivetyReportText.pack()
        naivetyReportText.insert(END, tempNaivetyReportString)

        scrollbar2Naive.config(command=naivetyReportText.xview)
        scrollbarNaive.config(command=naivetyReportText.yview)

    buttonWatchBeeMovements = Button(top, text="Watch Bee Movements",command=launchWatchBeeMovements)
    buttonWatchBeeMovements.grid(row=14,column=0,sticky=E)
    buttonViewTransitionMatrix = Button(top, text="View Transition Matrix",command=launchViewTransitionMatrix)
    buttonViewTransitionMatrix.grid(row=14,column=1,sticky=E)
    buttonViewIndividualPatchBreakdown = Button(top, text="View Individual Patch
Breakdown",command=launchViewIndividualPatchBreakdown)
    buttonViewIndividualPatchBreakdown.grid(row=14,column=2,sticky=E)
    buttonViewNaivetyReport = Button(top, text="View Naivety Report",command=launchViewNaivetyReport)
    buttonViewNaivetyReport.grid(row=15,column=2,sticky=E)
    infoLabel = Label(top, text="1) WATCH BEE MOVEMENTS will only show accurate patch\nlocations when patch
coordinate override not used",fg='blue')
    infoLabel.grid(row=15, column=0 ,sticky=W)
    infoLabel2 = Label(top, text="2) You MUST click on VIEW INDIVIDUAL PATCH BREAKDOWN to generate the new
individual patch breakdown file", fg='blue')
    infoLabel2.grid(row=16, column=0, sticky=W)

def main():
  GUIInput().mainloop()

if __name__ == "__main__":
  main()
```

Bibliography

Andersson M. (1978) Optimal foraging area : size and allocation of search effort. *Theoretical Population Ecology* **13**, 397-409

Bateman A. J. (1947) Contamination of seed crops. III Relation with isolation distance. *Heredity* **1**, 303-306

Beauchamp G. (2000) Learning rules for social foragers : implications for the producer-scrounger game and ideal free distribution theory. *Journal of Theoretical Biology* **207**, 21-35

Beecham J. A., Farnsworth K. D. (1998) Animal foraging from an individual perspective : an object oriented model. *Ecological Modelling* **113**, 141-156

Belcher K., Nolan J., Phillips P.W.B. (2005) Genetically modified crops and agricultural landscapes: spatial patterns of contamination. *Ecological Economics* **53**, 387-401.

Bell G. (1985) On the function of flowers. *Proceedings of the Royal Society of London. B* **224**, 223-265

Bernstein C., Kacelnik A., Krebs J. R. (1988) Individual decisions and the distribution of predators in a patchy environment. *Journal of Animal Ecology* **57**, 1007-1026

Best L. S., Bierzychudek P. (1982) Pollinator foraging on foxglove (digitalis purpurea): a test of a new model. *Evolution* **35**, 70-79

Biesmeijer J. C., Tóth E. (1998) Individual foraging, activity level and longevity in the stingless bee Melipona beecheii in Costa Rica (Hymenoptera, Apidae, Meliponinae). *Insectes Sociaux* **45**, 427-443

Birmingham A. L., Winston M. L. (2004) Orientation and drifting behaviour in bumblebees (Hymenoptera : Apidae) in commercial tomato greenhouses. *Can. J. Zool.* **82**, 52-59

Brian A. D. (1952) Division of labour and foraging in bombus agrorum Fabricius. *Journal of Animal Ecology* **21**, 223-240

Bullock D. S., Desquilbet M. (2002), The economics of non-GMO segregation and identity preservation. *Food Policy* **27**, 81-99

Burns J. G., Thomson J. D. (2006) A test of spatial memory and movement patterns of bumblebees at multiple spatial and temporal scales. *Behavioural Ecology* **17**, 48-55

Bush R.R., Mosteller F. (1951) A mathematical model for simple learning. *Psychological Review* **58**, 313-323

Campbell D.R. (1985) Pollen and gene dispersal: the influences of competition for pollination. *Evolution* **39**, 418-431

Carroll A. B., Pallardy S. G., Galen C. (2001) Drought stress, plant water status, and floral trait expression in fireweed, Epilobium angustifolium (Onagraceae). *American Journal of Botany* **88**, 438-446

Cartar R. V. (1991) A test of risk-sensitive foraging in wild bumble bees. *Ecology* **72**, 888-895

Cartar R. V. (1992) Morphological senescence and longevity: an experiment relating wing wear and life span in foraging wild bumble bees. *Journal of Animal Ecology* **61**, 225-231

Cartar R. V., Abrahams M. V. (1996), Risk-sensitive foraging in a patch departure context : a test with worker bumble bees. *Amer. Zool.* **36**, 447-458

Cartar R. V., Dill L. M. (1990) Colony energy requirements affect the foraging currency of bumble bees. *Behavioural Ecology and Sociobiology* **27**, 377-383

Cartar R.V. (1992) Adjustment of foraging effort and task switching in energy-manipulated wild bumblebee colonies. *Animal Behaviour* **44**, 75-88

Cartar R.V. (2004) Resource tracking by bumble bees: Responses to plant-level differences in quality. *Ecology* **85**, 2764-2771

Carvell C., Meek W. R., Pywell R. F., Goulson D., Nowakowski M. (2007) Comparing the efficacy of agri-environment schemes to enhance bumble bee abundance and diversity on arable field margins. *Journal of Applied Ecology* **44**, 29-40

Castellanos M. C., Wilson P., Thomson J. D. (2002) Dynamic nectar replenishment in flowers of Penstemon (Scrophulariaceae). *American Journal of Botany* **89**, 111-118

CEC (Commission of the European Communities) (2003) Regulation (EC) No 1830/2003 of the European Parliament and of the council of 22 September 2003 concerning the traceability and labelling of genetically modified organisms and the traceability of food and feed products produced from genetically modified organisms and amending Directive 2001/18/EC. Brussels, Belgium p. 27 (http://eur-lex.europa.eu/pri/en/oj/dat/2003/l_268/l_26820031018en00240028.pdf)

Charnov E. (1976) Optimal foraging : the marginal value theorem. *Theoretical Population Biology* **9**, 129-136

Charnov E.L. (1976) Optimal foraging: attack strategy of a mantid. *American Naturalist* **110**, 141-151

Chittka L., Leadbeater E. (2005) Social learning: Public information in insects. *Current Biology* **15**, R869-R871

Chittka L., Williams N. M., Rasmussen H., Thomson J. D. (1999) Navigation without vision: bumblebee orientation in complete darkness. *Proc. R. Soc. Lond. B* **266**, 45-50

Cibula D. A., Zimmerman M. (1984) The effect of plant density on departure decisions : testing the marginal value theorem using bumblebees and Deplhinium nelsonii. *Oikos* **43**, 154-158

Colbach N., Devaux C., Angevin F. (2009) Comparative study of the efficiency of buffer zones and harvest discarding on gene flow containment in oilseed rape. A modelling approach. *European Journal of Agronomy* **30**, 187-198

Colborn M., Ahmad-Annuar A., Fauria K., Collett T. S. (1999) Contextual modulation of visuomotor associations in bumble-bees (Bombus terrestris). *Proc. R. Soc. Lond.* **266**, 2413-2418

Conner A. J., Glare T. R., Nap J. (2003) The release of genetically modified crops into the environment: Part II overview of ecological risk assessment. *Plant Journal* **33**, 19-46

Cowie R. J. (1977) Optimal foraging in great tits (Parus major). *Nature* **268**, 137-139

Crane M. B., Mather K. (1943), The natural cross-pollination of crop plants with particular reference to the radish. *Annals of Applied Biology* **30**, 301-308

Cranmer L. (2004) The influence of linear landscape features on pollinator behaviour. PhD thesis, University of Leicester, Leicester, UK.

Cresswell J. (2006) Models of pollinator-mediated gene dispersal in plants. *Ecology and evolution of flowers* (eds L.D. Harder & S.C.H. Barrett), pp. 83-101 (Oxford University Press, Oxford, UK)

Cresswell J. E. (1990) How and why do nectar-foraging bumblebees initiate movements between inflorescences of wild bergamot Monarda fistulosa (Lamiaceae)?. *Oecologia* **82**, 450-460

Cresswell J. E., Hoyle M. (2006) A mathematical method for estimating patterns of flower-to-flower gene dispersal from a simple field experiment. *Function Ecology* **20**, 245-251

Cresswell J. E., Osborne J. L., Bell S. A. (2002) A model of pollinator-mediated gene flow between plant populations with numerical solutions for bumblebees pollinating oilseed rape. *Oikos* **98**, 375-384

Cresswell J. E., Osborne J. L., Goulson D. (2000), An economic model of the limits to foraging range in central place foragers with numerical solutions for bumblebees. *Ecological Entomology* **25**, 249-255

Cresswell J.E. (1997) Spatial heterogeneity, pollinator behaviour and pollinator-mediated gene flow: bumblebee movements in variously aggregated rows of oil-seed rape. *Oikos* **78**, 546-556

Cresswell, J.E. (1999) The influence of nectar and pollen availability on pollen transfer by individual flowers of oil-seed rape (*Brassica napus*) when pollinated by bumblebees (*Bombus lapidarius*). *Journal of Ecology* **87**, 670-677.

Dale P. J. *Where science fits into the GM debate. Gene flow from GM plants* (eds G.M. Poppy & M.J. Wilkinson), pp. 1-11. (Blackwell Publishing, Oxford, UK, 2005)

Dall S.R.X., Giraldeau L.A., Olsson O., McNamara J.M., Stephens D.W. (2005) Information and its use by animals in evolutionary ecology. *Trends In Ecology & Evolution* **20**, 187-193

Damgaard C., Kjellsson G. (2005) Gene flow of oilseed rape (Brassica napus) according to isolation distance and buffer zone. *Agriculture, Ecosystems and Environment* **108**, 291-301

Darvill B., Knight M. E., Goulson D. (2004) Use of genetic markers to quantify bumblebee foraging range and nest density. *Oikos* **107**, 471-478

Darwin C. (1859) On the origin of the species by means of natural selection or the preservation of favoured races in the struggle for life. John Murray, London.

Devlin B., Ellstrand N. C. (1990) Male and female fertility variation in wild radish, a hermaphrodite. *The American Naturalist* **136**, 87-107

Dingle H., Drake V. A. (2007) What is migration?. *Bioscience* **57**, 113-121

Dornhaus A., Chittka L. (2005) Bumble bees (Bombus terrestris) store both food and information in honeypots. *Behavioral Ecology* **16**, 661-666.

Dow S.M., Lea S.E.G. (1987) Sampling of schedule parameters by pigeons: tests of optimizing theory. *Animal Behaviour* **35**, 102-114.

Dramstad W. E. (1996) Do bumblebees (Hymenoptera : Apidae) really forage close to their nests?. *Journal of Insect Behaviour* **9**, 163-182

Dreisig H. (1995) Ideal Free Distributions of Nectar Foraging Bumblebees. *Oikos* **72**, 161-172

Dukas R., Real L. A. (1993) Effects of recent experience on foraging decisions by bumble bees. *Oecologia* **94**, 244-246

Dukas R., Real L.A. (1993) Effects of nectar variance on learning by bumble bees. *Animal Behaviour* **45**, 37-41.

Ellstrand N.C. (1992) Gene flow by pollen - implications for plant conservation genetics. *Oikos* **63**, 77-86.

Ellstrand N.C. (2001) When transgenes wander, should we worry? *Plant Physiology* **125**, 1543-1545.

Emlen J. M. (1966) The role of time and energy in food preference. *American Naturalist* **100**, 611-617

English-Loeb G. M., Karban R. (1992) Consequences of variation in flowering phenology for seed head herbivory and reproductive success in Erigeron glaucus (Compositae). *Oecologia* **89**, 588-595

Fauvergue X., Boll R., Rochat J., Wajnberg E., Bernstein C., Lapchin L. (2006) Habitat assessment by parasitoids: consequences for population distribution. *Behavioral Ecology* **17**, 522-531.

Fenster C.B. (1991) Gene flow in *Chamaecrista fasciculata* (Leguminosae) I. Gene dispersal. *Evolution* **45**, 398-409.

Ferrari M.J., Bjornstad O.N., Partain J.L., Antonovics J. (2006) A gravity model for the spread of a pollinator-borne plant pathogen. *American Naturalist* **168**, 294-303.

Firmage D. H., Cole F. R. (1988) Reproductive success and inflorescence size of calopogon tuberosus (orchidaceae). *Amer. J. Bot.* **75**, 1371-1377

Fretwell S. D., Lucas H. L. (1969) On territorial behavior and other factors influencing habitat distribution in birds. *Acta Biotheoretica* **19**, 16-36

Frisch K.V. *Bees: their vision, chemical senses and language* (Cornell University Press, Ithaca, NY, 1950)

Galen C., Cuba J. (2001) Down the tube: pollinators, predators, and the evolution of flower shape in the alpine skypilot, polemonium viscosum. *Evolution* **55**, 1963-1971

Galen C., Stanton M. L. (1989) Bumble bee pollination and floral morphology: factors influencing pollen dispersal in the alpine sky pilot, polemonium viscosum (polemoniaceae). *Amer. J. Bot.* **76**, 419-426

Gegear R. J., Thomson J. D. (2004), Does the flower constancy of bumble bees reflect foraging economics?. *Ethology* **110**, 793-805

Gilliam J. F., Fraser D. F. (1987) Habitat selection under predation hazard : test of a model with foraging minnows. *Ecology* **68**, 1856-1862

Giraldeau L-A., Kramer D. L. (1982) The marginal value theorem : a quantitative test using load size variation in a central place forager, the eastern chipmunk, Tamius striatus. *Animal Behaviour* **30**, 1036-1042

Goodell K., Elam D. R., Nason J. D., Ellstrand N. C. (1997) Gene flow among small populations of a self-incompatible plant: an interaction between demography and genetics. *American Journal of Botany* **84**, 1362-1371

Goss-Custard J. D., Caldow R. W. G., Clarke R. T., Durrell S., Sutherland W. J. (1995) Deriving population parameters from individual variations in foraging behaviour. 2. Model tests and population parameters. *Journal of Animal Ecology* **64**, 277-289

Goulson D. (1999) Foraging strategies of insects for gathering nectar and pollen, and implications for plant ecology and evolution. *Perspectives in Plant Ecology* **2**, 185-209

Goulson D. (2000) Why do pollinators visit proportionally fewer flowers in large patches?. *Oikos* **91**, 485-492

Goulson D. (2003) Effects of introduced bees on native ecosystemsol. *Annu. Rev. Ecol. Evol. Syst.* **34**, 1-26

Goulson D. *Bumblebees* (Oxford University Press, Oxford, UK, 2003)

Goulson D., Hanley M. E., Darvill B., Ellis J. S., Knight M. E. (2005) Causes of rarity in bumblebees. *Biological Conservation* **122**, 1-8

Goulson D., Lye G. C., Darvill B. (2008) Decline and conservation of bumble bees. *Annual Review of Entomology* **53**, 191-208

Goulson D., Stout J. (2001) Homing ability of the bumblebee Bombus terrestris (Hymenoptera : Apidae). *Apidologie* **32**, 105-111

Graham L., Jones K. N. (1996) Resource partitioning and per-flower foraging efficiency in two bumble bee species. *American Midland Naturalist* **136**, 401-406

Greenleaf S. S., Williams N. M., Winfree R., Kremen C. (2007) Bee foraging ranges and their relationship to body size. *Oecologia* **153**, 589-596

Grimm V., Railsback S. F. *Individual-based modelling and ecology* (Princeton University Press, 2005)

Groβ R., Houston A. I., Collins E. J., McNamara J. M., Dechaume-Moncharmont F., Franks N. R. (2008) Simple learning rules to cope with changing environments. *J. R. Soc. Interface* **5**, 1193-1202

Hancock P.A., Milner-Gulland E.J. (2006) Optimal movement strategies for social foragers in unpredictable environments. *Ecology* **87**, 2094-2102.

Harder L. D., Barrett S. C. H. *Ecology and evolution of flowers* (Oxford University Press, 2006)

Harder L. D., Real L. A. (1987) Why are bumble bees risk averse?. *Ecology* **68**, 1104-1108

Hassall M., Lane S.J. (2005) Partial feeding preferences and the profitability of winter-feeding sites for brent geese. *Basic And Applied Ecology* **6**, 559-570.

Hayter K., Cresswell J. (2006) The influence of pollinator abundance on the dynamics and efficiency of pollination in arable *Brassica napus*: implications for landscape-scale gene dispersal. *Journal of Applied Ecology* **43**, 1196-1202.

Heinrich B. (1976a) The foraging specializations of individual bumblebees. *Ecological Monographs* **46**, 105-128

Heinrich B. (1976b) Flowering phonologies : bog, woodland, and disturbed habitats. *Ecology* **57**, 890-899

Heinrich B. (1979) "Majoring" and "Minoring" by foraging bumblebees, bombus vagans: an experimental analysis. *Ecology* **60**, 246-255

Heinrich B. (1979) Resource heterogeneity and patterns of movement in foraging bumblebees. *Oecologia* **40**, 235-245

Heinrich B. (1983) Do bumblebees forage optimally, and does it matter? *American Zoologist* **23**, 273-281.

Heinrich B. (*Bumblebee economics* (Harvard University Press, Cambridge, Mass., 1979)

Heinrich B., Raven P. H. (1972), Energetics and pollination ecology. *Science* **176**, 597-602

Hirvonen H., Ranta E., Rita H., Peuhkuri N. (1999) Significance of memory properties in prey choice decisions. *Ecological Modelling* **115**, 177-189.

Hodges C. M. (1985a) Bumble bee foraging : energetic consequences of using a threshold departure rule. *Ecology* **66**, 188-197

Hodges C. M. (1985b) Bumble bee foraging : the threshold departure rule. *Ecology* **66**, 179-187

Holyoak M., Lawler S. P. (1996) The role of dispersal in predator-prey metapopulation dynamics. *Journal of Animal Ecology* **65**, 640-652

Horn M.E., Woodard S.L., Howard J.A. (2004) Plant molecular farming: systems and products. *Plant Cell Reports* **22**, 711-720

Houston A. I. (2009) Flying in the Face of Nature. *Behavioural Processes* **80**, 295-305

Hoyle M., Cresswell J. E. (2007) A search theory model of patch-to-patch forager movement with application to pollinator-mediated gene flow. *Journal of Theoretical Biology* **248**, 154-163

Hoyle M., Cresswell J. E. (2007a) The effect of wind direction on cross-pollination in wind-pollinated GM crops. *Ecological Applications* **17**, 1234-1243

Hoyle M., Hayter K., Cresswell J. E. (2007) Effect of pollinator abundance on self-fertilization and gene flow : application to GM canola. *Ecological Applications* **17**, 2123-2135

Inglis I.R., Langton S., Forkman B., Lazarus J. (2001) An information primacy model of exploratory and foraging behaviour. *Animal Behaviour* **62**, 543-557.

Ingram J. (2000) The separation distances required to ensure cross-pollination is below specified limits in non-seed crops of sugar beet, maize and oilseed rape. *Plant Varieties and Seeds* **13**, 181-199.

Inouye D. W. (1978) Resource partitioning in bumblebees: experimental studies of foraging behaviour. *Ecology* **59**, 672-678

Isnec M.R., Couvillon P.A., Bitterman M.E. (1997) Short-term spatial memory in honeybees. *Animal Learning & Behavior* **25**, 165-170.

Iwasa Y., Higashi M., Yamamura N. (1981) Prey distribution as a factor determining the choice of optimal foraging strategy. *The American Naturalist* **117**, 710-723

Johst K., Brandl R., Pfeifer R. (2001) Foraging in a patchy and dynamic landscape: Human land use and the White Stork. *Ecological Applications* **11**, 60-69.

Kacelnik A., Bateson M. (1996) Risky theories-the effects of variance on foraging decisions. *American Zoologist* **36**, 402-434

Kacelnik A., Krebs J.R. (1985). Learning to exploit patchily distributed food. In *Behavioural ecology (The 25th symposium of the British Ecological Society)* (eds R.M. Sibly & R.H. Smith), pp. 189-205. Blackwell Scientific Publications, Oxford.

Kamil A. C. (1983) Optimal foraging theory and the psychology of learning. *American Zoologist* 1983 **23**, 291-302

Keasar T., Rashkovich E., Cohen D., Shmida A. (2002) Bees in two-armed bandit situations : foraging choices and possible decision mechanisms. *Behavioral Ecology* **13**, 757-765

Kells A. R., Holland J. M., Goulson D. (2001) The value of uncropped field margins for foraging bumblebees. *Journal of Insect Conservation* **5**, 283-291

Klaassen R. H. G., Nolet B. A., van Gils J. A., Bauer S. (2006) Optimal movement between patches under incomplete information about the spatial distribution of food items. *Theoretical Population Biology* **70**, 452-463

Kleijnen J. P. C. (1995) Verification and validation of simulation models. *European Journal of Operational Research* **82**, 145-162

Klein A-M., Vaissière B. E., Cane J. H., Steffan-Dewenter I., Cunningham S. A., Kremen C., Tscharntke T. (2007) Importance of pollinators in changing landscapes for world crops. *Proc. R. Soc. B* **274**, 303-313

Klein E.K., Lavigne C., Picault H., Michel R., Gouyon P.H. (2006) Pollen dispersal of oilseed rape: estimation of the dispersal function and effects of field dimension. *Journal of Applied Ecology* **43**, 141-151.

Klinger T., Arriola P. E., Ellstrand N. C. (1992) Crop-weed hybridization in radish (Raphanus sativus): effects of distance and population size. *American Journal of Botany* **79**, 1431-1435

Klinkhamer P. G. L., de Jong T. J. (1990) Effects of plant size, plant density, and sex differential nectar reward on pollinator visitation in the protandrous Echium vulgare (Boraginaceae). *Oikos* **57**, 399-405

Knight M. E., Martin A. P., Bishop S., Osborne J. L., Hale R. J., Sanderson R. A., Goulson D. (2005) An interspecific comparison of foraging range and nest density of four bumblebee (Bombus) species. *Molecular Ecology* **14**, 1811-1820

Kohlmann S. G., Risenhoover K. L. (1998), Effects of resource distribution, patch spacing, and preharvest information on foraging decisions of northern bobwhites. *Behavioural Ecology* **9**, 177-186

Koltowski Z. (2002) Beekeeping value of recently cultivated winter rapeseed cultivars. *Journal of Apicultural Science* **46**, 23-32

Kotliar N. B., Wiens J A (1990), Multiple scales of patchiness and patch structure : A hierarchical framework for the study of heterogeneity. *Oikos* **59**, 253-260

Kramer P. J. (1950) Effects of wilting on the subsequent intake of water by plants. *American Journal of Botany* **37**, 280-284

Krebs J. R., Erichsen J. T., Webber M. I., Charnov E. L. (1977) Optimal prey selection by the Great Tit (Parus major). *Animal Behaviour* **25**, 30-38

Krebs J. R., Kacelnik A. (1984) Time horizons of foraging animals. *Annals of the New York Academy of Sciences*, **423**, 278-291

Krebs J. R., Kacelnik A., Taylor P. (1978) Test of optimal sampling by foraging great tits. *Nature* **275**, 27-31

Lea S. E. (1979) Foraging and reinforcement schedules in the pigeon : optimal and non-optimal aspects of choice. *Animal Behaviour* **27**, 875-886

Lilliendahl K. (1998) Yellowhammers get fatter in the presence of a predator. *Animal Behaviour* **55**, 1335-1340

Lima S. L. (1984) Downy woodpecker foraging behaviour : efficient sampling in simple stochastic environments. *Ecology* **65**, 166-174

Llewellyn D., Fitt G. (1996), Pollen dispersal from two field trials of transgenic cotton in the Namoi Valley, Australia. *Molecular Breeding* **2**, 157-166

Llewellyn D., Tyson C., Constable G., Duggan B., Beale S., Steel P. (2007) Containment of regulated genetically modified cotton in the field. *Agriculture, Ecosystems and Environment* **121**, 419-429

MacArthur R. H., Pianka E. R. (1966) On optimal use of a patchy environment. *American Naturalist* **100**, 603-609

Mangel M., Clark C. W. (1986) Towards a unified foraging theory. *Ecology* **67**, 1127-1138
Manning A. (1956) Some aspects of the foraging behaviour of bumble-bees. *Behaviour* **9**, 164-201

Mazur J. E. (1984) Tests of an equivalence rule for fixed and variable reinforcer delays. *Journal of Experimental Psychology* **10**, 426-436

McFrederick Q. S., LeBuhn G. (2006), Are urban parks refuges for bumble bee Bombus spp. (Hymenoptera : Apidae)?. *Biological Conservation* **129**, 372-382

McNamara J. M., Green R. F., Olsson O. (2006) Bayes' theorem and its applications in animal behaviour. *Oikos* **112**, 243-251

McNamara J.M., Houston A.I. (1987) Memory and the efficient use of information. *Journal of Theoretical Biology* **125**, 385-395.

Messeguer J., Marfà V., Català M. M., Guiderdoni E., Melé E. (2004) A field study of pollen-mediated gene flow from Mediterranean GM rice to conventional rice and the red rice weed. *Molecular Breeding* **13**, 103-112

Michener C. D. *The social behaviour of the bees* (Harvard University Press, 1974) 404pp

Milinski M. (1994) Long-term memory for food patches and implications for ideal free distributions in sticklebacks. *Ecology* **75**, 1150-1156.

Mitchell R. J. (1993) Adaptive significance of ipomopsis agrregata nectar production : observation and experiment in the field. *Evolution* **47**, 25-35

Mogensen V. O., Jensen C. R., Mortensen G., Thage J. H., Koribidis J., Ahmed A. (1996) Spectral reflectance index as an indicator of drought of field grown oilseed rape (Brassica napus L.). *European Journal of Agronomy* **5**, 125-135

Morris W. F., Kareiva P. M., Raymer P. L. (1994) Do barren zones and pollen traps reduce gene escape from transgenic crops?. *Ecological Applications* **4**, 157-165

Morse D. H. (1986) Predatory risk to insects foraging at flowers. *Oikos* **46**, 223-228

Nap J., Metz P. L. J., Escaler M., Conner A. J. (2003), The release of genetically modified crops into the environment : Part I Overview of current status and regulations. *The Plant Journal* **33**, 1-18

Neuman P., Ahearn W. H., Hineline P. N. (1997) Pigeons' choices between fixed-ratio and geometrically escalating schedules. *Journal of the Experimental Analysis of Behaviour* **68**, 357-374

Niv Y., Joel D., Meilijson I., Ruppin E. (2002) Evolution of reinforcement learning in uncertain environments : A simple explanation for complex foraging behaviours. *Adaptive Behaviour* **10**, 5-24

Oerke E-C., Dehne H-W. (1997) Global crop production and the efficacy of crop protection – current situation and future trends. *European Journal of Plant Pathology* **103**, 203-215

Ohashi K., Leslie A., Thomson J. D. (2008) Trapline foraging by bumble bees : V. Effects of experience and priority on competitive performance. *Behavioural Ecology* **19**, 936-948

Olsson O., Brown J.S. (2006) The foraging benefits of information and the penalty of ignorance. *Oikos* **112**, 260-273.

Osborne J. L., Clark S. J., Morris R. J., Williams I. H., Riley J. R., Smith A. D., Reynolds D. R., Edwards A. S. (1999), A landscape-scale study of bumble bee foraging range and constancy, using harmonic radar. *Journal of Applied Ecology* **36**, 519-533

Osborne J. L., Williams I. H. (2001) Site constancy of bumble bees in an experimentally patchy habitat. *Agriculture, Ecosystems and Environment* **83**, 129-141

Ott J. R., Real L. A., Silverfine E. M. (1985) The effect of nectar variance on bumblebee patterns of movement and potential gene dispersal. *Oikos* **45**, 333-340

Peat J., Goulson D. (2005) Effects of experience and weather on foraging rate and pollen versus nectar collection in the bumblebee. *Behav Ecol Sociobiol* **58**, 152-156

Pham-Delegue M. H., Bailez O., Blight M. M., Masson C., Picard-Nizou A. L., Wadhams L. J. (1993) Behavioural discrimination of oilseed rape volatiles by the honeybee Apis mellifera L. *Chemical Senses* **18**, 483-494

Pierce G. J., Ollason J. G. (1987) Eight reasons why optimal foraging theory is a complete waste of time. *Oikos* **49**, 111-117

Pierre J., Mesquida J., Marilleau R., Pham-Delègue M. H., Renard M. (1999) Nectar secretion in winter oilseed rape, Brassica napus – quantitative and qualitative variability among 71 genotypes. *Plant Breeding* **118**, 471-476

Pleasants J. M. (1981) Bumblebee response to variation in nectar availability. *Ecology* **62**, 1648-1661

Pleasants J. M. (1989) Optimal foraging by nectarivores : a test of the marginal-value theorem. *The American Naturalist* **134**, 51-71

Pleasants J. M., Chaplin S. J. (1983) Nectar production rates of Asclepias quadrifolia : causes and consequences of individual variation. *Oecologia* **59**, 232-238

Pleasants J. M., Zimmerman M. (1979) Patchiness in the dispersion of nectar resources : evidence for hot and cold spots. *Oecologia* **41**, 283-288

Pleasants J. M., Zimmerman M. (1983) The distribution of standing crop of nectar : what does it really tell us?. *Oecologia* **57**, 412-414

Plowright R. C., Galen C. (1985) Landmarks or obstacles: the effects of spatial heterogeneity on bumble bee foraging behaviour. *Oikos* **44**, 459-464

Poppy G., Wilkinson M. (eds) *Gene flow from GM plants.* (Blackwell Publishing, Oxford, 2005)

Proctor M., Yeo P., Lack A. *The natural history of pollination.* (Harper Collins, London, 1996)

Pyke G. H. (1978) Optimal foraging : movement patterns of bumblebees between inflorescences. *Theoretical Population Biology* **13**, 72-98

Pyke G. H. (1978) Optimal foraging in hummingbirds : testing the marginal value theorem. *American Zoologist* **18**, 739-752

Pyke G. H. (1984) Optimal foraging theory : a critical review. *Ann. Rev. Ecol. Syst.* **15**, 523-575

Pyke G. H., Cartar R. V. (1992) The flight directionality of bumblebees: do they remember where they came from?. *Oikos* **65**, 321-327

Pyke G.H. (1979) Optimal foraging in bumblebees: rules of movement between flowers within inflorescences. *Animal Behaviour* **27**, 1167-1181.

Rademaker M. C. J., De Jong T. J., Kilnkhamer P. G. L. (1997) Pollen dynamics of bumble-bee visitation on Echium vulgare. *Functional Ecology* **11**, 554-563

Railsback S.F., Lamberson R.H., Harvey B.C., Duffy W.E. (1999) Movement rules for individual-based models of stream fish. *Ecological Modelling* **123**, 73-89.

Ramsay G., Thomson C., Squire G. (2003) Quantifying landscape-scale gene flow in oilseed rape. *DEFRA Report*

Raney T. (2006) Economic impact of transgenic crops in developing countries. *Current Opinion in Biotechnology* **17**, 1-5

Rasheed S. A., Harder L. D. (1997) Economic motivation for plant species preferences of pollen-collecting bumble bees. *Ecological Entomology* **22**, 209-219

Real L. A. (1981) Uncertainty and pollinator-plant interactions : the foraging behaviour of bees and wasps on artificial flowers. *Ecology* **62**, 20-26

Real L. A., Rathcke B. J. (1991) Individual variation in nectar production and its effect on fitness in Kalmia Latifolia. *Ecology* **72**, 149-155

Reboud X. (2003), Effect of a gap on gene flow between otherwise adjacent transgenic Brassica napus crops. *Theoretical and Applied Genetics* **106**, 1048-1058

Ricketts T. H. (2001) The matrix matters : effective isolation in fragmented landscapes. *The American Naturalist* **158**, 87-99

Rieger M. A., Lamond M., Preston C., Powles S. B., Roush R. T. (2002) Pollen-mediated movement of herbicide resistance between commercial canola fields. *Science* **296**, 2386-2388

Riley J. R., Reynolds D. R., Smith A. D., Edwards A. S., Osborne J. L., Williams I. H., McCartney H. A. (1999) Compensation for wind drift by bumble bees. *Nature* **400**, 126

Robertson A. W., Mountjoy C., Faulkner B. E., Roberts M. V., Macnair M. R. (1999) Bumble bee selection of mimulus guttatus flowers : the effects of pollen quality and reward depletion. *Ecology* **80**, 2594-2606

Robledo-Arnuncio J.J., Austerlitz F. (2006) Pollen dispersal in spatially aggregated populations. *American Naturalist* **168**, 500-511.

Rodd F. H., Plowright R. C., Owen R. E. (1980) Mortality rates of adult bumble bee workers (Hymenoptera : Apidae). *Can. J. Zool.* **58**, 1718-1721

Rodriguez-Gironés M., Vásquez R. A. (1997) Density-Dependent Patch Exploitation and Acquisition of Environmental Information. *Theoretical Population Biology* **52**, 32-42

Roschewitz I., Thies C., Tscharntke T. (2005) Are landscape complexity and farm specialisation related to land-use intensity of annual crop fields?. *Agriculture, Ecosystems and Environment* **105**, 87-99

Roubik D. W., Buchmann S. L. (1984) Nectar selection by Melipona and Apis mellifera (Hymenoptera : Apidae) and the ecology of nectar intake by bee colonies in a tropical forest. *Oecologia* (Berlin) **61**, 1-10

Rubetra S. (1981) Central place foraging in the whinchat. *Ecology* **62**, 538-544

Saville N. M., Dramstad W. E., Fry G. L. A., Corbet S. A. (1997) Bumblebee movement in a fragmented agricultural landscape. *Agriculture, Ecosystems and Environment* **61**, 145-154

Schmitt J. (1983) Density-dependent pollinator foraging, flowering phenology, and temporal pollen dispersal patterns in Linanthus bicolor. *Evolution* **37**, 1247-1257

Schoen D. J., Clegg M. T. (1985) The influence of flower color on outcrossing rate and male reproductive success in ipomoea purpurea. *Evolution* **39**, 1242-1249

Schoener T.W. (1971) Theory of feeding strategies. *Annual Review of Ecology and Systematics* **11**, 369-404.

Skinner B. F. *The Behavior of Organisms: An Experimental Analysis*. (Cambridge, Massachusetts: B. F. Skinner Foundation, 1938)

Slatkin M. (1985) Gene flow in natural populations. *Annual Review of Ecology and Systematics* **16**, 393-430.

Smith J.N.M., Dawkins R. (1971) Hunting Behavior Of Individual Great Tits In Relation To Spatial Variations In Their Food Density. *Animal Behaviour* **19**, 695-706.

Stephens D. W., Krebs J. R. (1986) *Foraging Theory* (Princeton University Press, 1986)

Strickler E., Vinson J.W. (2000) Simulation of the effect of pollinator movement on alfalfa seed set. *Environmental Entomology* **29**, 907-918.

Sutton R. S., Barto A. G. *Reinforcement Learning* (MIT Press, 1999)

Suzuki Y., Kawaguchi L. G., Toquenaga Y. (2007) Estimating nest locations of bumblebees Bombus ardens from flower quality and distribution. *Ecol Res* **22**, 220-227

Thomson J. D. (1988) Effects of variation in inflorescence size and floral rewards on the visitation rates of traplining pollinators of Aralia hispida. *Evolutionary Ecology* **2**, 65-76

Thomson J. D. (1996) Trapline foraging by bumblebees: I. Persistence of flight-path geometry. *Behavioural Ecology* **7**, 158-164

Thomson J. D., Peterson S. C., Harder L. D. (1987) Response of traplining bumble bees to competition experiments: shifts in feeding location and efficiency. *Oecologia* **71**, 295-300

Thuijsman F., Peleg B., Amitai M., Shmida A. (1995) Automata, Matching And Foraging Behavior Of Bees. *Journal of Theoretical Biology* **175**, 305-316.

Timmons A. M., O'Brien E. T., Charters Y. M., Dubbels S. J., Wilkinson M. J. (1995) Assessing the risks of wind pollination from fields of genetically modified Brassica napus ssp. oleifera. *Euphytica* **85**, 417-423

Tinbergen N., Impekoven M., Franck D. (1967) An experiment on spacing out as a defence against predators. *Behaviour* **28**, 307-321.

Todd P. M. (2000) The Ecological Rationality of Mechanisms Evolved to Make Up Minds. *Animal Behavioral Scientist* **43**, 940-956

Tyler J. A., Hargrove W. W. (1997) Predicting spatial distribution of foragers over large resource landscapes: A modeling analysis of the Ideal Free Distribution. *Oikos* **79**, 376-386.
Valone T.J. (2006) Are animals capable of Bayesian updating? An empirical review. *Oikos* **112**, 252-259.

van der Veen I. T. (1999) Effects of predation risk on diurnal mass dynamics and foraging routines of yellowhammers (Emberiza citrinella). *Behavioural Ecology* **10**, 545-551

Visscher P.K., Seeley T.D. (1982) Foraging strategy of honeybee colonies in a temperate deciduous forest. *Ecology* **63**, 1790-1801.

Waddington K. D., Allen T., Heinrich B. (1981) Floral preferences of bumblebees (Bombus edwardsii) in relation to intermittent versus continuous rewards. *Animal Behaviour* **29**, 779-784

Walklate P. J., Hunt J. C. R., Higson H. L., Sweet J. B. (2004) A model of pollinator-mediated gene flow for oilseed rape. *Proc. R. Soc. Lond. B* **271**, 441-449

Ward J. F., Austin R. M., MacDonald D. W. (2000) A simulation model of foraging behaviour and the effect of predation risk. *Journal of Animal Ecology* **69**, 16-30

Waser P. M. (1985) Does competition drive dispersal?. *Ecology* **66**, 1170-1175

Wehner R., Michel B., Antonsen P. (1996) Visual navigation in insects: coupling of egocentric and geocentric information. *The Journal of Experimental Biology* **199**, 129-140

Werner E. E., Gilliam J. F., Hall D. J., Mittelbach G. G. (1983) An experimental test of the effects of predation risk on habitat use in fish. *Ecology* **64**, 1540-1548

Werner E.E., Mittelbach G.G. (1981) Optimal foraging: field tests of diet choice and habitat switching. *American Zoologist* **21**, 813-829.

Westphal C., Steffan-Dewenter I., Tscharntke T. (2003), Mass flowering crops enhance pollinator densities at a landscape scale. *Ecology Letters* **6**, 961-965

Westphal C., Steffan-Dewenter I., Tscharntke T. (2006), Bumblebees experience landscapes at different spatial scales : possible implications for coexistence. *Oecologia* **149**, 289-300

Westphal C., Steffan-Dewenter I., Tscharntke T. (2006), Foraging trip duration of bumblebees in relation to landscape-wide resource availability. *Ecological Entomology* **31**, 389-394

Wilkinson M.J., Elliott L.J., Allainguillaume J., Shaw M.W., Norris C., Welters R., Alexander M., Sweet J. & Mason D.C. (2003) Hybridization between Brassica napus and B. rapa on a national scale in the United Kingdom. *Science* **302**, 457-459

Williams N. (2008), Bee fears heighten. *Current Biology* **18**, 682-683

Williams N. M., Thomson J. D. (1998) Trapling foraging by bumble bees: III. Temporal patterns of visitation and foraging success at single plants. *Behavioural Ecology* **9**, 612-621

Williams P. H., Osborne J. L. (2009) Bumblebee vulnerability and conservation world-wide. *Apidologie* **40**, 367-387

Wyatt R., Broyles S. B., Derda G. S. (1992) Environmental influences on nectar production in milkweeds (asclepias syriaca and a. exaltata). *American Journal of Botany* **79**, 636-642

Ydenberg R. C. (1984) Great tits and giving-up times : decision rules for leaving patches. *Behaviour* **90**, 1-24

Young H. J., Stanton M. L. (1990) Influences of floral variation on pollen removal and seed production in wild radish. *Ecology* **71**, 536-547

Zimmerman M. (1979) Optimal foraging : a case for random movement. *Oecologia* **43**, 261-267

Zimmerman M. (1981) Optimal foraging, plant density and the marginal value theorem. *Oecologia* **49**, 148-153