The University of Birmingham
School of Computer Science
MSc in Advanced Computer Science

Mini-project

Evolving Visual Tracking using Genetic Programming

John S. Montgomery
msc37jxm@cs.bham.ac.uk
Supervisor: Dr J.L. Wyatt

April 25, 2004

Abstract

The lure of Genetic Programming is the promise of automatically generating "programs". I demonstrate the use of my C++ based genetic programming framework to evolve programs for visual tracking. In particular I evolve programs using both "Minimal Simulations" and real data to detect the position *and* velocity of objects in motion.

For each experiment several different parameters sets are used, mainly testing the effect of different mutation and recombination rates. It was generally found that higher mutation rates gave better results, but this apparent improvement was not always statistically significant.

I also successfully demonstrate the transfer of two programs, evolved by Minimal Simulation, to an application that provides them with live data from a webcam. This application displays the programs output overlaid on the input image and displayed. The programs are seen to perform well at detecting the on-screen position and velocity of single objects moving against a relatively uniform background - the task for which they were evolved.

Keywords

Genetic Programming, Co-evolution, Minimal Simulation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Genetic Programming

1.1.1 Tree-based GP

Kozas [6] genetic programming is based on lisp s-expressions, which are essentially tree [1] structures. Rather than allowing the evolution of arbitrary programs, which might create programs that could actually do damage to the host computer (by accident) and also slow down evolution, Koza specifies a "function" set and a "terminal" set. The programs only contain nodes selected from both these sets. The terminal set consists of functions/inputs that have an "arity" (number of arguments) of zero, they therefore form the terminal nodes of a tree. The function set consists of those functions which have a non-zero arity, which therefore form branches in the program tree (see Figure 1.1).

Choosing a suitable function and terminal set is very important, as it directly affects how well suited a program can be to its task. Care must also be taken to gracefully handle *all* inputs to a given function. In short their must be "closure"

[1] A tree is a graph without circuits [15]

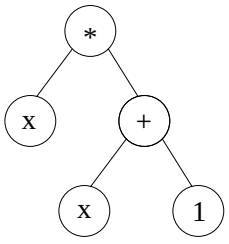

Figure 1.1: (* x (+ x 1))

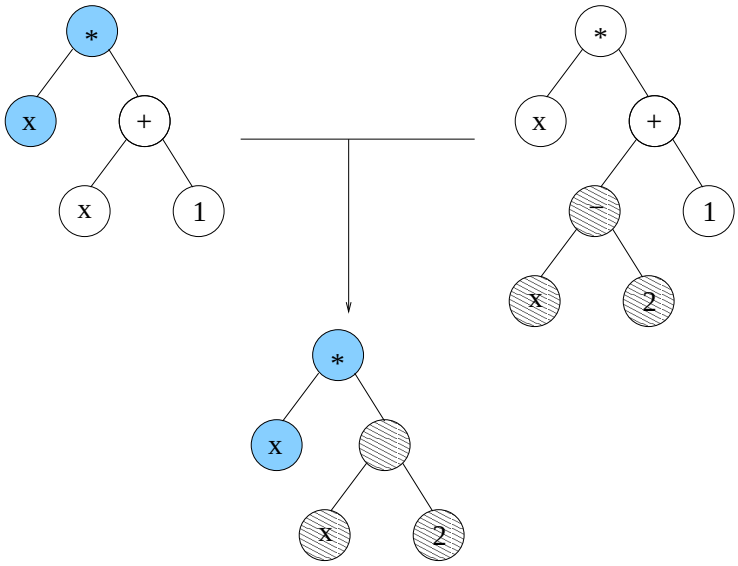

Figure 1.2: Subtree Swapping Recombination

of the function and terminal sets [6]. For example it would be no good if an evolutionary run was aborted, because a program happened to divide a number by zero. As an example standard division is usually replaced with a "protected" division operator [6]. If the divisor is zero with a protected division then the result will not be infinite [2], but will be 1.

As well as handling all inputs, the function and terminal sets must also be "sufficient" [6] for the problem at hand. It must be at the very least possible to form a correct/appropriate solution from the building blocks provided.

Koza proposed several operators to alter the structure of the programs during evolution. The two main ones being:

- Recombination - using sub-tree swapping (Figure 1.2).
- Mutation - by randomly re-initialising subtrees (Figure 1.3).

Koza also talks about several other operators (Permutation, Editing and Encapsulation), but these were not always used by him in his experiments. Koza also believes more in recombination than in mutation, because "... it is relatively rare for a particular function or terminal ever to disappear entirely from a population ..." [6]. Thus, in Kozas view, the need of mutation to restore "lost diversity" [6] is simply not needed.

[2] From a mathematical sense the result would be infinite, but on a computer with limited precision the result will either be "very large" or an error value of some description

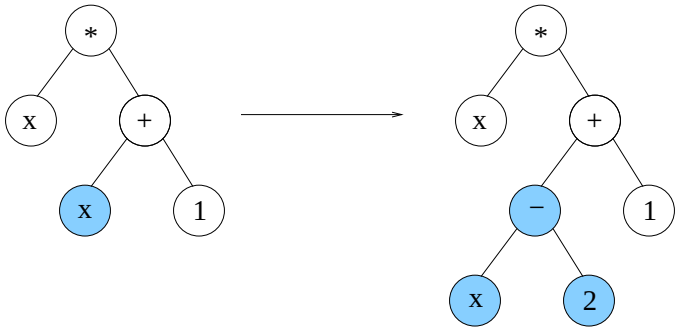

Figure 1.3: Subtree Mutation

Automatically Defined Functions

As most programmers know modularisation is generally a good thing. Without it programmers would constantly spend time either copying and pasting code, or else rewriting the same things over and over. By creating modular blocks that evolution can re-use we can hopefully speed up evolution, by letting it not have to constantly re-invent the wheel.

Koza proposed Automatically Defined Functions [6] (ADFs) as a way to allow evolved programs to make use of modularised code. An Automatically Defined Function (ADF) is simply a function tree, but one that is available as a function in the function set of other trees. Whereas previously there would merely have been one function tree now there are several. One of these trees is then the conventional result producing branch [14]/main function, with the other trees made available for evolution to use as it wishes. To avoid recursion (and therefore inifinte loops) an ADF does not use itself in its function set and a hierarchy of which ADFs can use which other ADFs must also be imposed.

Due to the different function sets of the ADFs crossover must also be altered, so that it only performs crossovers between counter-part ADFs. It would no good crossing-over the main function tree with a function tree that can be used by the main function - there would be a good chance we ended up with a recursive function call appearing after a while.

The draw back with ADFs is that the user must specify how many ADFs there will be for a given run and how many (and possibly what) arguments they require.

ADFs are probably the most flexible of the methods proposed for modularisation by Koza. As they are able to use different inputs they can be used in several different places, in a program and can therefore compute different (but related) values. One of the simplest alternative is providing a “Set function” [6]. This simply allows a program to store a computed value for later re-use. The “Set function” produces a side effect, which is to change the value of a given variable, so its position in the tree and when it is executed can potentially alter the final result in quite dramatic ways.

1.1.2 Alternative forms Genetic Programming

Tree-based GP is one of the first (if not *the* first) forms of Genetic Programming. However it has some drawbacks, noticeably that is often slow running (as traditionally Lisp has been used as the implementation language) and more particularly usually does not handle iteration and recursion very well.

Obviously speed is not a real issue, because implementations can be optimised for speed and memory usage and computers are getting faster. However iteration and recursion are more fundamental problems. In a naive implementation an infinite loop, created by accident, could wreak havoc as the entire system would be stuck in a loop and the run would need to be aborted. There are ways around this, but a lot of effort has gone into examining other forms of evolving programs by natural selection

Linear Genomes

Instead of using a tree-based approach to GP many people have tried using linear sequences of machine-code-like instructions. Typically the instructions are designed to operate on a virtual machine[11] of some sort, but some have even tried evolving actual machine code [14].

Graph Genomes

Trees are just a special type of (undirected) graph [15] that do not have cycles. So an obvious extension from using trees for genetic programming is "the more general approach of using graph structured representations" [8].

Whilst some approaches (e.g. TAG [10]) use acyclic (directed) graphs to avoid the issue of recursion, but still allow a certain amount of code re-use, others "positively embrace" recursion and iteration [14]. A classic example of this is PADO [13].

Graphs in PADO represent how control in a program flows. Each node in the graph performs some operation using a stack, where they retrieve their inputs and then return their results. Once they have performed their action they use a "branch-decision" function to decide which of the other nodes, that they are connected to, will be executed next. This approach obviously makes recursion and iteration very easy, but in a way that is much easy to monitor.

1.2 Visual Tracking

Visual Tracking "... is a basic skill in most visually equipped animals" [10]. In order to perform this task two basic behaviours [12] are required and conversely usually observed:

- Saccade
- Pursuit

Saccades are the rapid and sudden eye movements that occur when we change our attention to a different point in space (darting eyes if you will). Due to the fact that saccades are so rapid ("... a few tens of milliseconds" [9]) they can be a problem in computer vision, as the images captured in that period are "blurred because of the fast camera motion" [9]. Systems often initiate a saccade and then wait for it to finish before doing anything else. This can potentially result in problems if further saccades are required due to object motion, so some have done work on improving the ways saccades are handled, by also considering the object motion as a predictor [9].

Pursuit is the type of eye motion that occurs when the eye follows an object moving at a constant and therefore predictable, velocity. It is used to "keep the image of a moving target on the fovea footnoteCenter/focal point of the retina" [12].

Murray et al [5] have demonstrated a successful system that uses two processes running in parallel - one uses coarse images to "direct saccadic shifts of attention" [5] and the other uses a finer resolution towards the centre to handle pursuit. Both behaviours are used to successfully track a moving object. A saccade initially occurs to bring the object towards the centre of the view, then a "gaze controller" [5] hands over control to the pursuit behaviour. If the object gets out of view again the gaze controller returns control to the saccading behaviour.

Clearly detecting position *and* motion are essential characteristics of a successful visual tracking system.

1.3 Genetic Programming for Visual Tracking

1.3.1 TAG and Robot Shaping

In their paper, Perkins and Hayes [10] discuss the use of a genetic programming system called TAG to evolve controllers for visual tracking. Programs in TAG are *acyclic* graphs. The use of acyclic graphs guarantees that the programs will be recursion free, but still fosters a certain amount of re-use, as different branches may share intermediate results.

Perkins and Hayes attempt to "bring the human back into the design cycle" [10], by breaking down a training task into smaller sub-tasks that can be learnt incrementally and later combined. In this way they hope to be able to tackle more complex tasks than are currently possible. They refer to this process as "Robot Shaping" [10].

Minimal Simulations

A "Minimal Simulator" [2] [3] [4] was used for training the programs. A minimal simulation does not strive to be accurate, but instead attempts to model those features that are important for the desired behaviour. This will of course result in differences between the simulation and real life, so "mechanisms are put in place that prevent controllers from evolving to rely on them" [3]. This usual is

handled by adding sufficient noise, to inaccurately modeled behaviours, so as to render them unreliable for use by the controllers. Of course this means that one must also decide before hand which aspects of the simulation must be modeled accurately and which must not be used by evolution, however they also have the advantage of being very easy to implement and very fast running.

Perkins and Hayes used minimal simulators that modeled an object moving in front of a camera. In one version of the simulator the object was significantly brighter than the background (the "light tracking task" [10]), in another version the object was coloured and patterned in a similar fashion to the background. In the latter case the object position could only be reasonably determined by considering the change in pixel values in successive frames, which is to say only the objects motion could be used as a reliable feature for evolution - not its colour or apparent shape.

Chapter 2

Methodology

2.1 Tree Based Genetic Programming

As a way for the author to familiarise himself with Genetic Programming, the decision was made to implement a tree-based genetic programming system. As tree based GP is the "classic" form of GP it was felt that this would give a good introduction to some of the issues in the area. It would also help allow a better understanding of the advantages and limitations of this approach, so as to better be able to contrast it with other approaches.

2.1.1 C++ Templates

The core genetic programming code was written using C++ templates. This allows (more or less) what is known as "parametric polymorphism" [1], which is to say that the type of data that the evolved programs deal with can easily be changed. The data type to be used must be specified at compile time, for efficiency reasons, but even this allows a great deal of flexibility. For example early versions of the code could be tested with simple integers or real numbers, before more specific code needed to be written. In particular some simple "symbolic regressions" [6] were performed, to make sure that the system was at least able to cope with these simple tasks.

2.1.2 Steady State GA

The algorithm used was a "Steady State GA". After generating an initialise population we proceed to iterate the algorithm in the following fashion:

- Independently select two parents via tournament [2] selection.
- With a certain probability create a child by recombining the two parents; otherwise by cloning one of the parents.

[1] http://en.wikipedia.org/wiki/Polymorphism_(computer_science)#Parametric_polymorphism
[2] Normally a binary tournament, but possibly with more individuals.

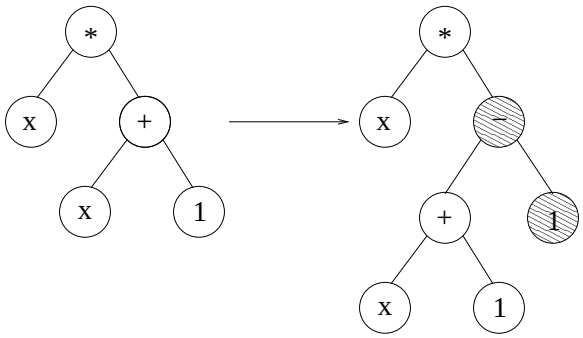

Figure 2.1: Parent/Root Mutation

- With a certain probability mutate the child.
- Select a random individual from the population.
- Replace that individual with the child only if the child is better or else with a small probability [3].

Recombination

A standard subtree-swapping recombination [14] operator was used. However if the swap points selected would create a tree that exceeded a certain depth then new swap points would be selected, up to five times, until a valid pair of points could be used. If no such points were found then the recombination simply did nothing (resulting in a clone of one parent).

Mutation

Several different mutation operators were implemented:

- subtree mutation - a random subtree is selected and it is re-initialised (randomly) to form a new subtree.
- collapse subtree mutation - a random subtree is selected and replaced with a randomly chosen terminal.
- point mutation - select a random *node* and change its function (randomly) to one of the same type/arity.
- "parent/root" mutation - select a random node, then either replace it with one of its child nodes or replace it with a new node that uses the original node as one of it's children (see Figure 2.1).

[3] e.g. $p = 0.02$.

The first three mutation operators are standard [14] operators, but the fourth - "parent/root" mutation - is slightly different. It is (at least partially) similar in intent to the "hoist" mutation operator. The hoist operator simply selects a random subtree and makes it the the root of the entire tree [14]. The idea behind the "parent/root" operator is to move subtrees up and down the hierarchy, not necessarily straight to the top as with hoist.

All of the mutation operators were implemented so that they preserve the maximum depth of the trees. This helps avoid trees that are excessively large in size and therefore helps us to preserve some semblance of speed.

2.2 The OpenCV Image Processing Library

The OpenCV [1] library is an open source computer vision library available via IBM. It can be used under Linux and Windows. The main advantage of using the library is that it provides a large number of very powerful computer vision algorithms, that have typically been highly optimised. It obviously makes sense not to re-invent the wheel when it comes to these algorithms. There are also facilities for loading and displaying images. The library functions that are used by the function set are listed in Appendix C.

2.3 ImageArray Values

The data type used for the image processing work was a class called "ImageArray". The class provided overloaded assignment and copy constructors so it could be used in a similar fashion to the builtin types, such as "int" and "double", meaning it can be used in the templated GP code mentioned previously. Each ImageArray object has a specific "type":

- scalar - a simple double precision real number.
- vector - a two dimensional vector/coordinate (two double precision real numbers).
- image - a two dimensional array of (real-valued) pixels stored as an IplImage pointer from the OpenCV library. [4]

As the ImageArray class would often be copied and deleted throughout the course of executing a program tree, care was taken to minimise unnecesary memory allocation and deallocation. To this end a memory pool [7] was maintained of image objects, which could be quickly recycled for new ImageArray objects. The memory pooling combined with minimising the total number number of ImageArray objects active at any one time helped to ensure a generally low memory footprint. Typically this was about 8Mb - with a population size

[4] Images are grey scale light intensity values, for the sake of simplicity, speed and memory usage.

of 1000 programs. By comparison an earlier version of the code, that maintained one preallocated ImageArray (and associated image) for each node in each tree, used something approaching 1Gb! Conversions Sometimes it is necesary to convert between the different data types. The following conversions can be performed:

- image to scalar - by taking the average pixel value.
- scalar to image - by creating an image with all pixels set to the scalar value (using cvSet).
- image to vector - by taking the coordinates of the brightest pixel in the image.
- scalar to vector - by creating a vector with x and y set to the scalar value.
- vector to scalar - by taking the magnitude of the vector.

The conversion from a vector to an image is never performed.

2.4 Functions and Terminals

2.4.1 Terminals

Terminal functions have an arity of zero - they have no input, but have an output. There are two different types of terminals used:

- scalar constants (e.g. 0, 1, -1)
- input variables (e.g. the current image to be considered for processing).

A user defined set of constants are made available at the beginning of a run. They are never altered during a run, but can obviously be combined to "manufacture" other constants. Each program tree in the program has its own set of (independent) input variables. In a similar fashion to the arguments to a function in a normal programming language these are local to the tree.

2.4.2 Unary Functions

Unary functions take one input and produce an output. In the same manner as Perkins[10], if the input does not match the desired type, the output is simple set to be the input, i.e. the function does nothing with the input, except passing it on unaltered. This means if a unary operator is being used contrary to its design it will merely contribute to an increase in program size and not produce any unintentional side effects. Table 2.1 outlines the unary functions used.

Function	Input type	Output type	Description
avg	image	scalar	average pixel value [a]
peak	image	vector	coordinates of "highest" pixel [b]
cen	image	vector	centre of mass of the image
gauss	image	image	gaussian blur of the image
erode	image	image	"erode" the image
dilate	image	image	"dilate" the image
abs	any	as input	absolute value of input
neg	any	as input	negative version of input

[a] equivalent to an image to scalar conversion
[b] equivalent to an image to vector conversion

Table 2.1: Unary Functions

2.4.3 Binary Functions

Binary functions take two inputs and produce an output. Depending on the types of the inputs (scalar, vector or image) conversions are performed so that the inputs can be "meaningfully" used. The ordering of the inputs matters for some functions (subtraction, division), so at the conversion stage we must assume the ordering matters. This means that without conversion there would be nine different case to deal with. To make matters worse not all of these variations can be dealt with sensibly. For example an image and a scalar can be easily combined, forming another image, but adding a two dimensional vector to an image makes little sense. So the inputs to a binary function are first queried for their types and then converted as appropriate:

- If the the types are equal no conversion is performed, and the output type will match the input type.
- If one input is an image and the other a scalar then no conversion [5] is performed, and the output type will be an image.
- If either input is a vector then the other input is converted to a vector and the output will also be a vector.

In most cases OpenCV [1] provides functions for combining images and scalars (e.g. cvAddS, cvSubS), but where no equivalent is provided we obtain image and use cvSet(scalar) to create an image with all pixels equal to the scalar and then use the relevant image based functions.
The binary functions and how they operate on the different data-types are outlined in Table 2.2.

[5] Where possible

Function	Image-Image	Image-Scalar	Vector-Vector [a]	Scalar-Scalar
+	cvAdd	cvAddS	addition	addition
−	cvSub	cvSubS/cvSubRS	subtraction	subtraction
*	cvMul	convert scalar	multiplication	multiplication
/ [b]	cvDiv	convert scalar	division	division
<	cvCmp	cvCmpS	less than	less than
>	cvCmp	cvCmpS	greater than	greater than
max	cvMax	cvMaxS	max value	max value
min	cvMin	cvMinS	min value	min value
diff	cvAbsDiff	cvAbsDiffS	absolute diff	absolute diff

[a] equivalent to Scalar-Scalar operations, but performed on the vectors x and y values
[b] division is "protected"

Table 2.2: Binary Functions

2.4.4 Ternary Functions

Currently there is only one ternary function: the canny edge detection function. This takes an input image and two scalar parameters. The first input must be an image, otherwise the output is set to be the first input (in the same fashion as the unary operators). The other two inputs, are converted to scalars and used as the two thresholds required by the canny function.

2.4.5 Automatically Defined Functions

Every function tree in a program is considered an ADF, with a simple hierarchy set up between them. As an example, most of the time, there are two function trees fn0 and fn1, with fn1 available as a function for use in fn0, but not the otherway around. The simple hierarachy means we do not have to worry about recursion and therefore inifinite loops.
Figure 2.2 shows an example program, consisting of two defined functions. As can be seen fn0 makes use of fn1, but not the other-way around, therefore no recursion will occur when the program is executed.

2.5 Visual Tracking

2.5.1 Minimal Simulations/Synthetic Images

A simple simulator was created for evolving programs that exhibit useful behaviour for visual tracking. The simulator was modelled approximately along the lines of a minimal simulation [2], but does not really meet all of the appropriate criteria. Essentially we are evolving using a set of synthetic images, which whilst not perfect does allow us to check that things are working as they are supposed to be. Within a relatively limited sense it does still produce programs that are a viable starting point for more robust visual tracking behaviour.

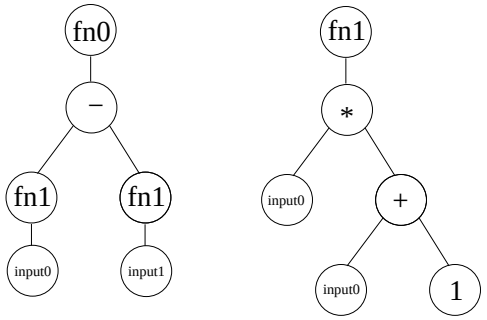

Figure 2.2: Example of Defined Functions in use

In a similar fashion to Perkins and Hayes [10] the minimal simulation represents an "object" on a "background" in two dimensions. I chose to use the objects colour as the significant attribute that the controllers should evolve to use. In particular the difference between the objects colour and the background colour is very important. Figure 2.3 shows a sample frame from the simulation, in this case being used for training to detect the objects position. As can been seen both the object and background are quite distinct, but contain a certain amount of noise, so overaly simplistic solutions will usually end up being slightly off centre.

Several instances of the minimal simulation are used for training at any one time, with each one having different parameters. The implementation of the minimal simulations is such that they always produce the same output, thus making comparison between competing programs - at a given instance - deterministic. However to avoid getting trapped with a population of overly adapted, but under generalised programs , we re-initialise the simulations after a given period - typically a generation [6].

2.5.2 Shaping in Parallel

As a twist on "Robot Shaping" [10] I elected to evolve using two separate tasks at the same time - with each task being assigned to a separate function. By evolving the tasks on separate functions we are then later able to make use of both behaviours later in the final controller.

Motion Task

The main task was to find the the objects velocity (or change in position) between two frames of input. In the minimal simulation the "object" was moved too random positions within the input field and fn0 (the first function) was

[6] As the algorithm is a steady state GA one generation is equivalent to generating a "populations worth" of children.

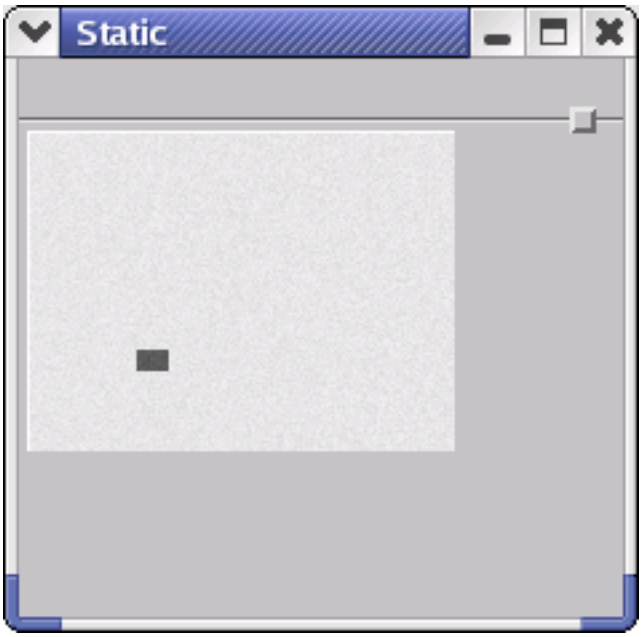

Figure 2.3: Minimal Sim Sample Frame

evaluated for this task. The second function (fn1) was made available to fn0 as an automatically defined function.

Position Task

The secondary task was to find the central position (on screen) of the object in any one given from of input. This task was evaluated using fn1 and only made use of the first frame of the motion task. As the object and background colours differed sufficiently for both tasks it meant that evolving a successful solution to the position task would also be very useful for the motion task.

Chapter 3

Experiments and Results

Four experiments, two using minimal simulations and two using real data, were performed. Each experiment consisted of five runs, each run five times (25 sets of results in total). The default parameters for the runs were as follow:

- Maximum Tree Depth = 6
- Iterations = 50000
- Population Size = 1000
- Tournament Size = 2
- Probability of Replacement = 0.0
- Probability of Recombination = 0.9
- Probability of Mutation = 0.01
  - Proportion of Subtree Mutations = 1.0
  - Proportion of Collapse Mutations = 0.0
  - Proportion of Root/Parent Mutations = 0.0
  - Proportion of Point Mutations = 0.0

The terminal sets [1] were also the same for each run:

- fn0 used `{ -1, 0, 1, input0, input1 }`
- fn1 used `{ -1, 0, 1, input0 }`

The five runs alter the parameters in the following ways:

- Run 1 Classic GP parameters
  - uses defaults.

[1] The input terminals are different for each function

- Run 2 Extra Mutation Operators
  - Proportion of Subtree Mutations = 1.0
  - Proportion of Collapse Mutations = 1.0
  - Proportion of Root Mutations = 1.0
  - Proportion of Point Mutations = 1.0
- Random Replacement
  - Probability of Replacement = 0.02
- Higher Mutation
  - Probability of Mutation = 0.1
- Half Recombination - Half Mutation
  - Probability of Recombination = 0.5
  - Probability of Mutation = 0.5

Program Listings

Evolved programs are listed in mock Lisp:

- all operators are prefix.
- brackets denote nesting.
- functions are defined using the word "def" followed by the function name, then the names of the inputs in brackets.

Internally the programs are actually C++ tree structures, but have been output in a Lisp style as Lisp is traditionally used in tree-based GP.

Statistical Analysis

A standard Mann-Whitney test was used to compare the results between runs. The Mann-Whitney test was used, because of the small sample sizes. It was applied only to the error rates and not the program sizes, because in most runs the program sizes were not being selected for until much later - if at all - so this would skew the results. The U values calculated give an indication of whether one sample is "better" than another. U values of 12.5 or mean that the samples are identical - lower means that the first sample is better and higher means it is worse. U values of 0 and 1 or 24 and 25 mean that there is a "significant" difference.

3.1 Minimal Simulations

Two different sets of runs were performed using a minimal simulation for evaluation. The difference between them being the nature of the operators provided. In the first set the gauss, erode and dilate operators were not used, but were used in the second set.

Additional Parameters

The following parameters were set the same for both experiments:

- Number of Test Cases = 5
- Epoch Length [2] = 1000
- Expected Error [3] = 16

Comparison of Final Results

To make comparison of results possible, between runs, after the main algorithm had run the entire population is re-evaluated on a much larger number of test cases (50 to be precise) that have been generated from a fixed random number seed. The large number of test cases should help us determine whether the programs have generalised well or not.

3.1.1 Basic Functions

This experiment used the function set:
`{ +, -, *, /, <, >, max, min, diff, neg, avg, cen, peak }`
The intention of this experiment was to get a baseline for how well evolution would manage with only very basic functions. The most complex function present is "cen" [4], but this was felt necessary as otherwise finding the centre of an object would be very tricky.

Results

Table 3.1 shows the results of all five runs. Figures 3.1 - 3.5 show how the error rates and program sizes have varied throughout the course of each run.

Even without recourse to statistical methods it is apparent that only three programs (out of twenty five) have been produced that approach the desired error rate of 16 sq pixels. Also no program has been produced that lies within the desired error rate. This means that no pressure has been applied to create smaller programs, so the program sizes shown have not been selected for at all - the programs evolved have only been selected for error rate and not program

[2] How many iterations before re-randomising the Test Cases
[3] Programs within this threshold will be selected for program size
[4] Center of Mass

Run 1		Run 2		Run 3		Run 4		Run 5	
Error	Size	Error	Size	Error	Size	Error	Size	Error	Size
1673.68	68	1446.88	47	1636.05	53	21.6867	15	1613	37
1655.75	45	1619.24	93	1701.33	54	1629.22	94	21.6796	24
1671.52	78	1632.15	54	1111.77	48	1209.42	73	800.013	55
826.89	76	1716.29	55	2656.67	23	2427.06	26	1488.72	44
1565.66	31	21.6122	15	1259.22	38	1739.01	12	1539.43	27

Table 3.1: Basic Functions Results

	Run 1	Run 2	Run 3	Run 4	Run 5
Run 1	12.5	10	14	13	4
Run 2	15	12.5	15	15	8
Run 3	11	10	12.5	11	6
Run 4	12	10	14	12.5	7
Run 5	21	17	19	18	12.5

Table 3.2: Basic Functions Mann-Whitney U Values

size. This is mannifested by the fact that Figures 3.1 - 3.5 show the program size tending to increase overtime.

Table 3.2 shows the Mann-Whitney U values for the error rates of the runs. As can be seen no one run is significantly different from any other, but Run 5 does overall have lower U values than the other runs - it would seem better, but maybe not significantly so.

Example Program Listing

This program was from Run 5 and had a final error rate of roughly 22 sq pixels and a size of 24. It contains some statements that are not necessary, but is otherwise approaching a good solution. Its main fault is that fn1, the position detector, does not attempt to find the centre of the object, but instead settles for the (approximate) top left hand corner of it.

```
# function 0 contains some extraneous expressions
def fn0 ( input0 input1 )
( +
 # this avg will do nothing, as it is passed a vector
 ( avg
  ( -
   ( diff
    ( fn1 input1 )
    ( min 0 ( peak input1 ) ) # vector (0,0)
   )
   ( * ( fn1 input0 ) 1 ) # unnecessary multiplication
  )
```

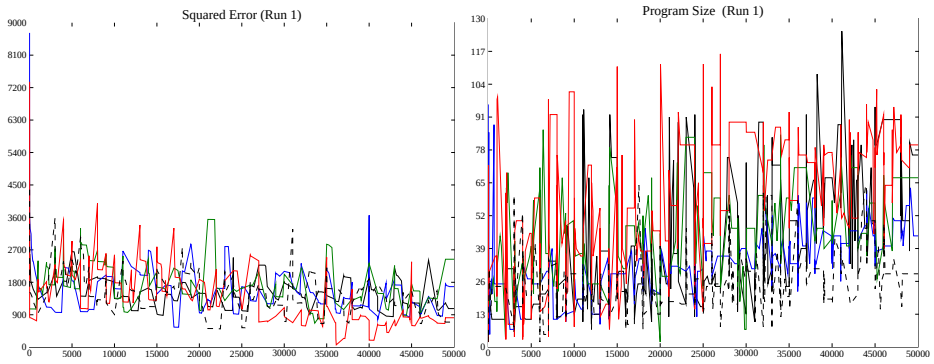

Figure 3.1: Basic Functions Run 1: Classic GP parameters.

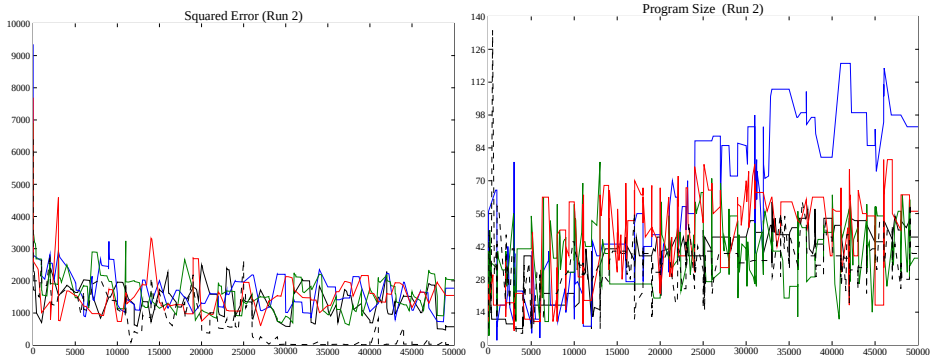

Figure 3.2: Basic Functions Run 2: Extra Mutation Operators.

```
 )
 ( avg ( <  ( / 0 0 ) input1 ) ) # scalar < 1
)

# function 1 is not quite exactly correct,
# as it does not find the objects centre
def fn1 ( input0 )
( diff
 ( avg input0 )
 input0
)
```

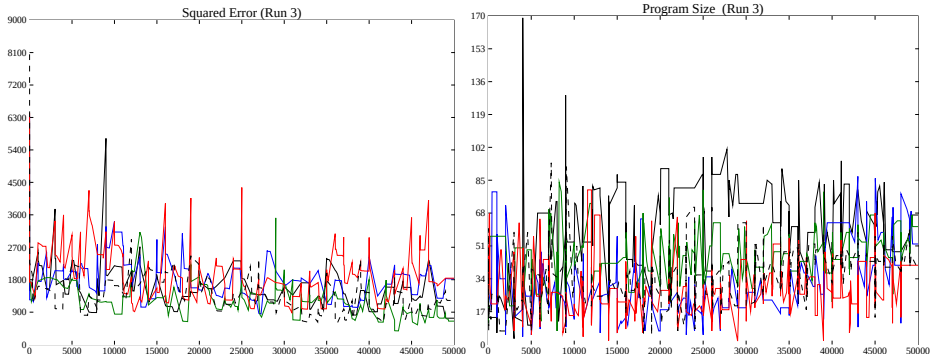

Figure 3.3: Basic Functions Run 3: Random Replacement.

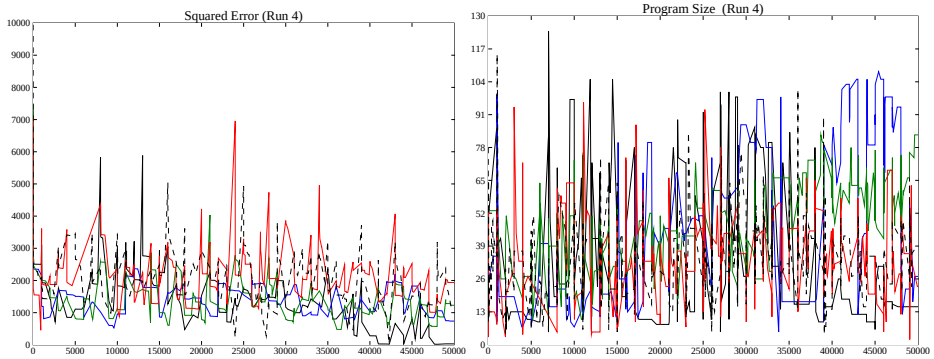

Figure 3.4: Basic Functions Run 4: Higher Mutation.

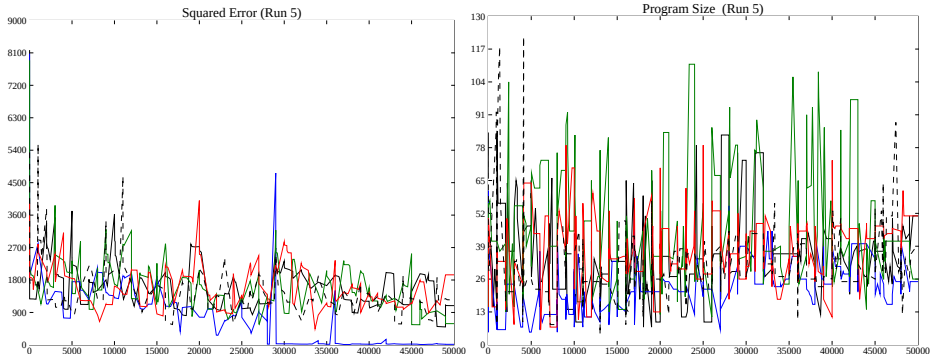

Figure 3.5: Basic Functions Run 5: Half Recombination - Half Mutation.

Run 1		Run 2		Run 3		Run 4		Run 5	
Error	Size	Error	Size	Error	Size	Error	Size	Error	Size
1711.17	33	1032.36	34	816.06	36	1538.49	43	1580.86	53
1700.76	57	1727.98	13	1301.74	59	1595.25	51	3.71667	18
1334.1	35	1606.56	75	1729.52	11	1672.12	26	1487.91	53
1653.53	39	1427.48	97	3.30333	11	1643.22	30	3.23	22
1649.9	57	1533.89	39	0.83	32	1703	30	7.57333	17

Table 3.3: Extra Functions Results

3.1.2 Extra Functions

This experiment used the function set:
`{ +, -, *, /, <, >, max, min, diff, neg, avg, cen, peak, gauss, erode, dilate }`
The intention being to see what effect adding the 3 new functions (gauss, erode and dilate) would have on the speed/effectiveness of evolution.

Results

Table 3.3 shows the results of all five runs. Figures 3.6 - 3.10 show how the error rates and program sizes have varied throughout the course of each run.

By comparison with the previous experiments (see Section 3.1.1) the results for this experiment are markedly better. Not only do we have five programs that are worthy of interest, but they are all also within the desired error rate of 16 sq pixels. It would appear that those extra functions may well have made a difference.

Table 3.4 shows the Mann-Whitney U values for this experiment. As opposed to the previous experiment there is one significantly different run. Run 5 has a U value of 1 when compared with Run 4. This means that the probability of Run 4 and Run 5 being the same is only 0.05. Interestingly Run 5 is very similar to Run 3, by U value, yet Run 3 is not significantly better than Run 4.

There is a slight problem with how the Mann-Whitney test is being applied in this case, as some of the error rates fall below the threshold. When this happens the error rates should really be considered equal when ranking, this might have yielded slightly different results overall, but only between those runs that have results below the threshold. This means that the comparison between Run 1 and 4 still holds, as no results from Run 4 fall below the threshold.

A simple comparison was also carried out between the results for this experiment and the previous one. This yielded of U value of 274 for the two sample of 25 results each. This proves not to be statistically significant, so despite the fact that there are qualitative differences between the results - i.e. no results in the previous experiment were below the threshold - there are real quantitative differences.

	Run 1	Run 2	Run 3	Run 4	Run 5
Run 1	12.5	8	5	10	2
Run 2	17	12.5	6	18	5
Run 3	20	19	12.5	20	13
Run 4	15	7	5	12.5	1
Run 5	23	20	12	24	12.5

Table 3.4: Extra Functions Mann-Whitney U Values

Example Program Listing

This program had a final error rate of about 8 sq pixels and a size of 17. Of particular interest, is that fn0 uses fn1 to devastating effect to make it very small indeed - also helped by the fact that fn1 forces its output to be a vector. As this program is within the desired error rate it has most likely benefited from at least some time for evolution to work at reducing program size.

```
# function 0 is very concise and near "perfect"
def fn0 ( input0 input1 )
( -
( fn1 input1)
( fn1 ( - ( avg -1 ) input0 ) )
)

# function 1 is near "perfect" too.
# also noteworthy as it makes use of gauss
# and erode functions which were unavailable
# in the previous runs
def fn1 ( input0 )
( peak # converts to vector, which is needed by fn0
 ( diff
  ( gauss ( avg input0 ) )
  ( erode ( erode ( gauss input0 ) ) )
 )
)
```

3.1.3 Minimal Simulation Reality Transfer

Figures 3.11 and 3.12 demonstrates two programs in action on a real webcam. They were loaded into a simple "test-harness" program from text files and then fed sequential frames from the webcam. Effectively from the programs point of view everything was the same as when they were running in simulation.

Two show the programs output a single line was drawn. The line starts according to the result of fn1 (the function that determines position) and proceeds in a direction according to the result of fn0 (the function that determines velocity).

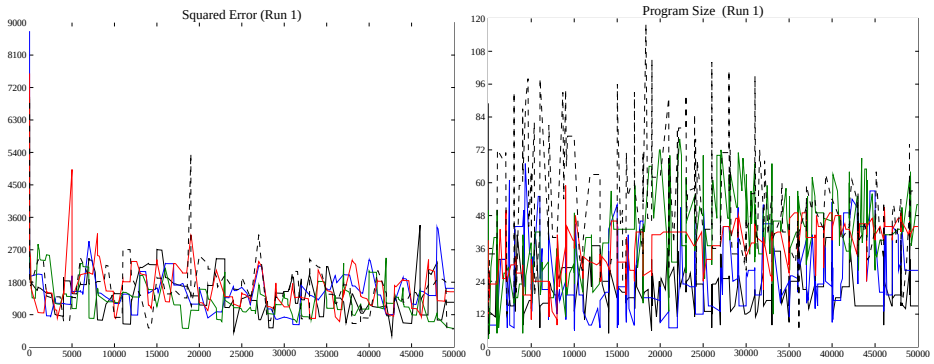

Figure 3.6: Extra Functions Run 1: Classic GP parameters.

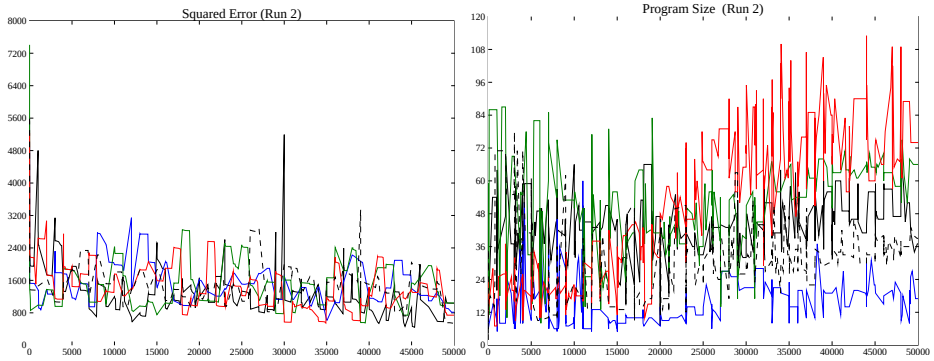

Figure 3.7: Extra Functions Run 2: Extra Mutation Operators.

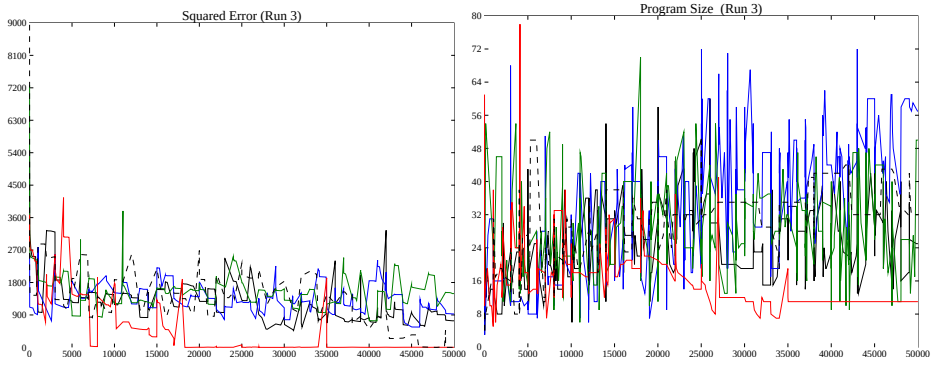

Figure 3.8: Extra Functions Run 3: Random Replacement.

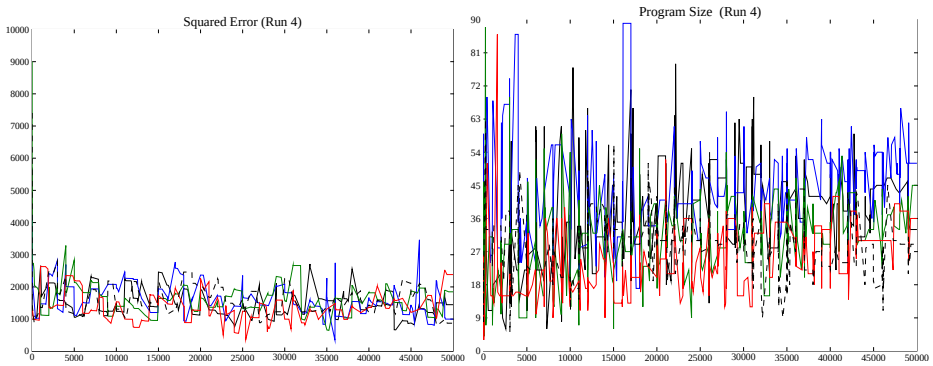

Figure 3.9: Extra Functions Run 4: Higher Mutation.

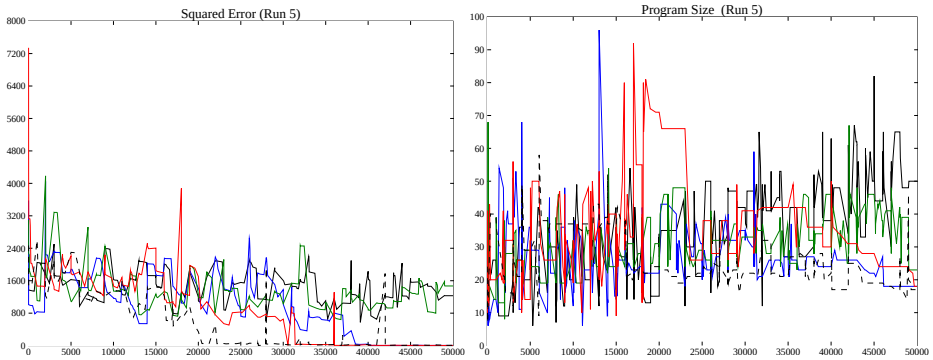

Figure 3.10: Extra Functions Run 5: Half Recombination - Half Mutation.

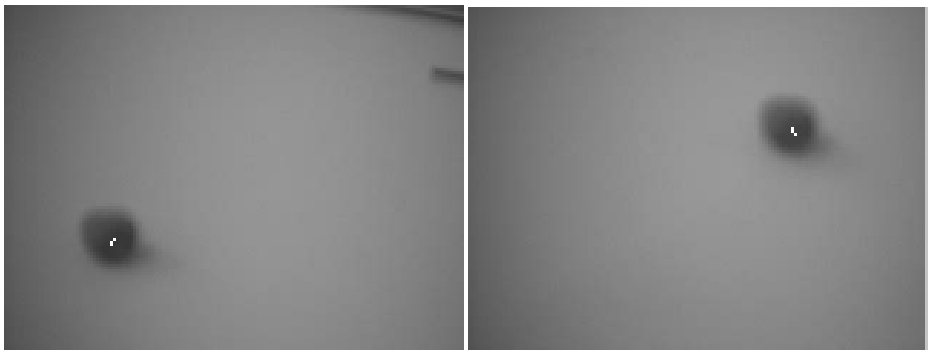

Figure 3.11: Static Camera Test.

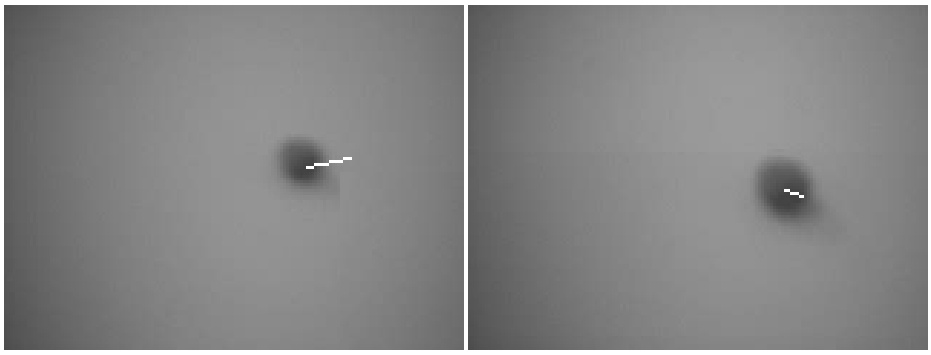

Figure 3.12: Moving Camera Test.

Figure 3.11 shows the programs output when the camera is kept still relative to the object [5]. As can be seen the line is positioned on top of the object in both cases, but does indicate a small amount of velocity. One presumes this small perceived velocity is due to noise - it may be that not enough noise was used when training the programs. This small amount of velocity did change direction a fair bit when the camera was kept still, so it would probably tend to cancel itself out overall.

Figire 3.12 shows the programs in action when the camera was moving relative to the object (thus simulating the object itself moving). The lines in this case are much longer and do indicate correctly the velocity [6] of the object. Although this is not readily apparent from a static picture.

Obviously these tests are fairly contrived, but they are appropriate considering the nature of the simulations used. With a cluttered view the programs simply do not know which thing is the object (the item worthy of its interest). With these uncluttered tests we can at least see that the simulations are not fundamentally flawed, they obviously simulate realty "well enough" for the programs to cope with the same situation in reality, as in the simulation.

[5] In this case a lump of blu-tac on a tabletop
[6] Relative to the camera

3.2 Real Data

Two different experiments were performed using a realdata set. The data consisted of several frames from a movie of a person hitting a ping-pong ball, available from:
`http://sampl.eng.ohio-state.edu/~sampl/data/motion/tennis/index.htm`
The images were converted to grayscale and rescaled to 217x148 pixels. The centre of the ping-pong ball was recorded for each frame to be used in training. The full set consists of a lot of large camera movements and times when the ping-pong ball disappears from view entirely, so only a subset of the images were actually used in both experiments.
The expected error rate for these experiments was set to 4 sq pixels for this run, as the size of the ping-pong ball was much smaller than the objects used in the minimal simulations. Also both experiments used the following function set:
This experiment used the function set:

```
{ +, -, *, /, <, >, max, min, diff, neg, avg, cen, peak,
gauss, erode, dilate, canny }
```

The canny operator was included to see if it had any effect on the results of the 2nd experiment.

3.2.1 Static Camera

Figure 3.13 shows two sample frames used for training during this experiment. They are quite representative of the total set. The ping-pong ball is seen to bounce up and down in the air several times, during the course of the frames.

Results

Table 3.5 shows the results of all five runs. Figures 3.14 - 3.18 show how the error rates and program sizes have varied throughout the course of each run.

In many ways this experiment is interesting, because the a good solution is usually found quickly we see that graphs for error in Figures 3.14 - 3.18 quickly fall - so much so that they look practically empty. This means that the real action occurs in the program size graphs. In most of these we see an initial overall increase in program size (particularly Runs 4 and 5 in Figures 3.17 and 3.18), as evolution is selecting for error. Then once the error rate falls below the threshold (of 4 sq pixels), we see a dramatic decline in program size, as selection changes to an interest in program size. As this often happens quite early in the runs we tend to up with very small programs, as seen in Table 3.5.

Table 3.6 shows the Mann-Whitney U values for the results. This shows that Runs 4 and 5 are significantly better than Run 1. The main difference between Run 4 and 5 versus Run 1 is the increase in mutation rate. So clearly for this experiment this was a deciding factor.

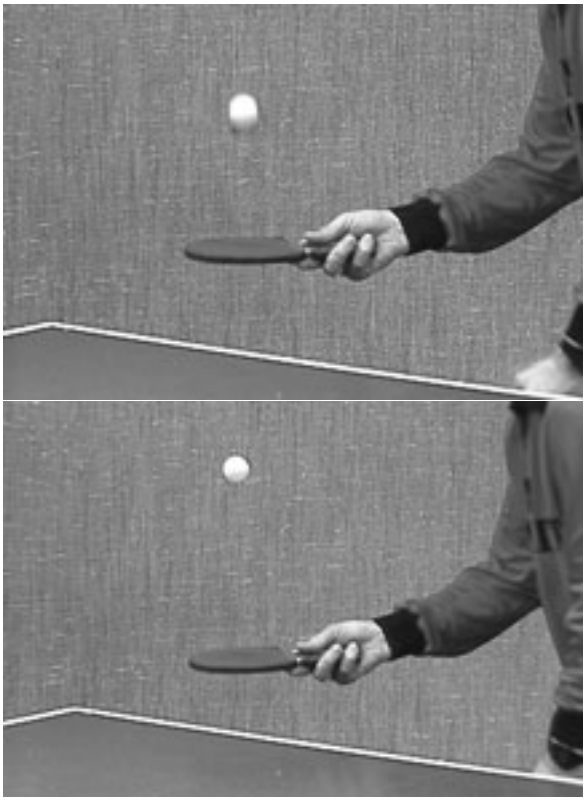

Figure 3.13: Example Movie Images (Static Camera)

Run 1		Run 2		Run 3		Run 4		Run 5	
Error	Size	Error	Size	Error	Size	Error	Size	Error	Size
57.3906	35	58.9375	59	2.79688	7	2.79688	6	2.79688	6
41.6719	81	3.09375	8	1.76562	8	3.60938	11	2.79688	6
48.7969	64	46.467	86	61.5495	44	2.79688	6	2.79688	6
57.2812	37	33.2188	26	23.9798	66	2.79688	6	2.79688	6
53.7344	48	2.79688	6	57.1814	32	34.1406	45	2.79688	6

Table 3.5: Static Camera Results

	Run 1	Run 2	Run 3	Run 4	Run 5
Run 1	12.5	6	8	0	0
Run 2	19	12.5	11.5	6.5	2.5
Run 3	17	13.5	12.5	9.5	7.5
Run 4	25	18.5	15.5	12.5	7.5
Run 5	25	22.5	17.5	17.5	12.5

Table 3.6: Static Camera Mann-Whitney U Values

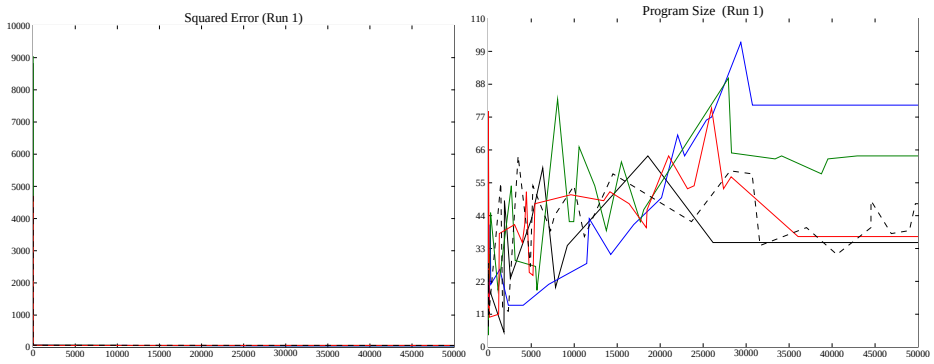

Figure 3.14: Static Camera Run 1: Classic GP parameters.

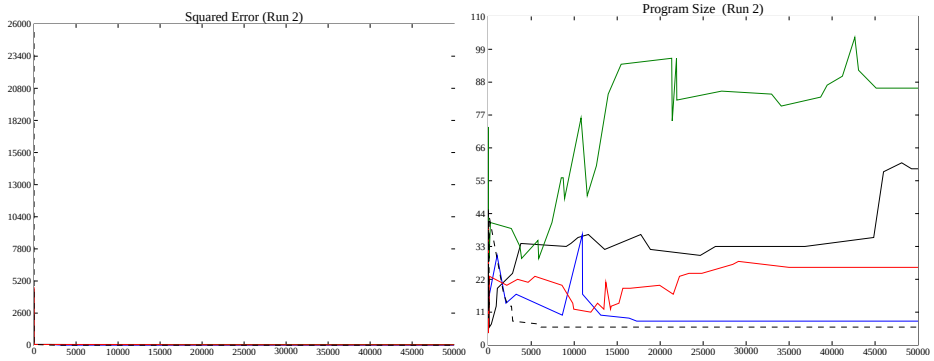

Figure 3.15: Static Camera Run 2: Extra Mutation Operators.

Example Program Listing

This is, at least for the given data, about as good one could hope. At 6 nodes in size, with an error rate of about 3 sq pixels, it probably could not be any smaller and still remain within the desired error rate of 4 sq pixels.

```
def fn0 ( input0 input1 )
( -
 ( peak input1 )
 input0
)

def fn1 ( input0 )
( gauss input0 )
```

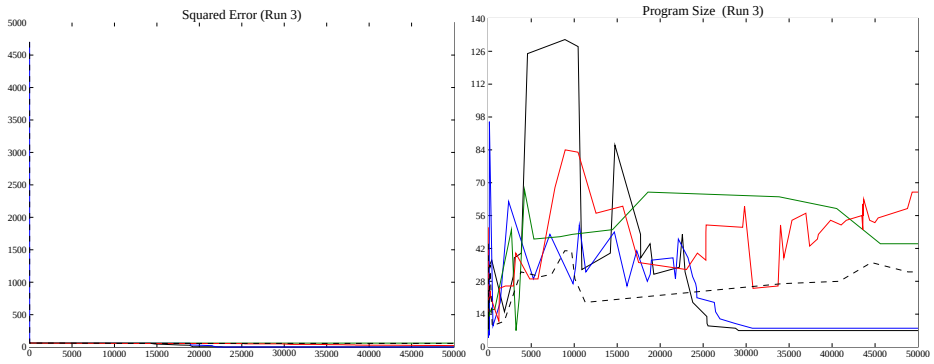

Figure 3.16: Static Camera Run 3: Random Replacement.

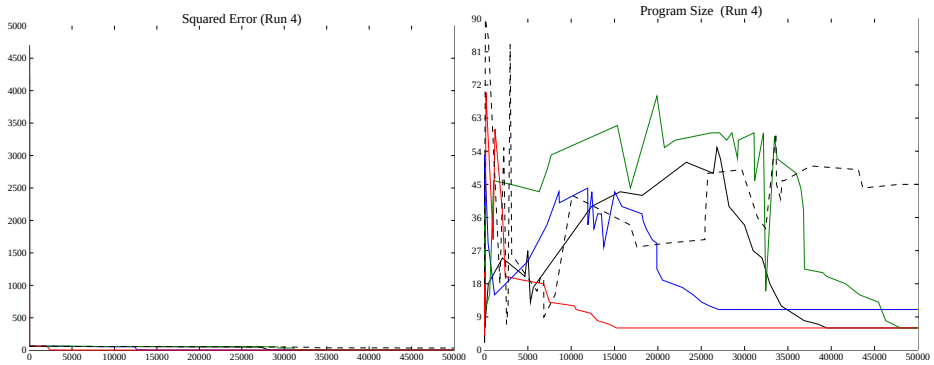

Figure 3.17: Static Camera Run 4: Higher Mutation.

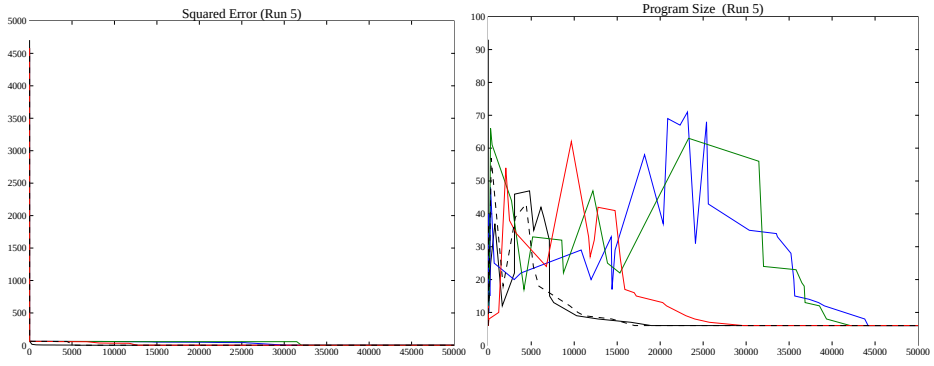

Figure 3.18: Static Camera Run 5: Half Recombination - Half Mutation.

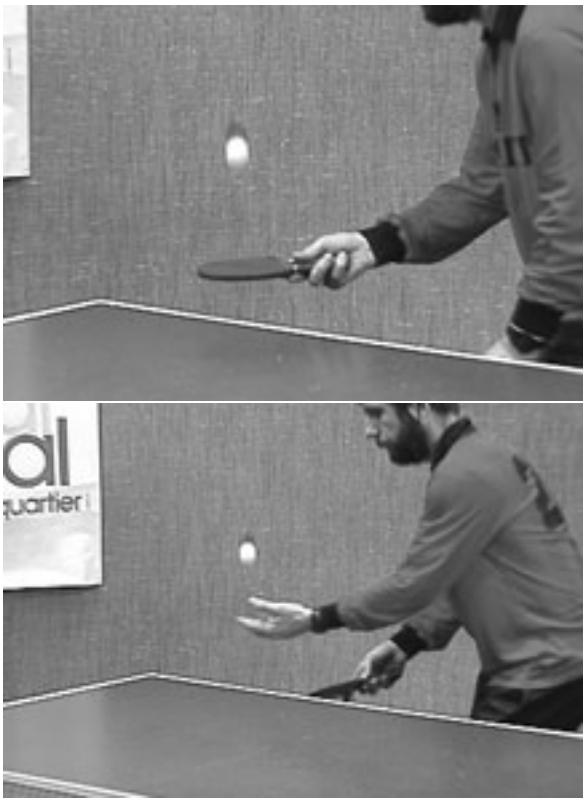

Figure 3.19: Example Movie Images (Moving Camera)

3.2.2 Moving Camera

Figure 3.19 shows sample frames from the complete sequence used for training. By comparison with the previous experiment (Section 3.2.1) the frames are qualitatively different. The frames shown are from the extra frames that this experiment used and show that the camera moving. This has a couple of effects. The first being that comparing the difference between successive frames is more difficult, as so much more has changed (at the pixel level). The second being that a large white poster comes into view at the top left of the frame. Because most of the operators used revolve around colour this presents a problem, because it may prove too distracting when searching for the ping-pong ball. This was one of the reasons for including the canny edge operator in the function set. Hopefully this would allow a bit more differentiation - although how exactly is not obvious.

Run 1		Run 2		Run 3		Run 4		Run 5	
Error	Size	Error	Size	Error	Size	Error	Size	Error	Size
245.974	37	116.047	68	511.057	97	262.823	55	91.783	78
98.6058	40	631.981	77	305.024	57	378.113	88	520.019	41
109.69	62	356.292	115	159.123	53	385.783	65	89.6132	39
240.636	48	419.415	50	302.981	65	189.498	56	91.0189	45
387.573	69	349.644	62	255.495	55	291.406	70	45.2736	60

Table 3.7: Moving Camera Results

	Run 1	Run 2	Run 3	Run 4	Run 5
Run 1	12.5	20	19	18	5
Run 2	5	12.5	8	9	4
Run 3	6	17	12.5	13	5
Run 4	7	16	12	12.5	5
Run 5	20	21	20	20	12.5

Table 3.8: Moving Camera Mann-Whitney U Values

Results

Table 3.7 shows the results of all five runs. Figures 3.20 - 3.24 show how the error rates and program sizes have varied throughout the course of each run.

As can be seen from the results (Table 3.7) were not overly impressive. It appears the moving camera and the extra clutter - particularly the appearance of the large white poster - were simply too much for evolution to handle, at least with the operators provided. I suspect that some sort of object shape or size recognition operators would be required to detect the object of interest (the ping-pong ball) and from their track its velocity. Although there is the possibility that the hard limit on the program tree depth might need to be relaxed a bit too.

Table 3.8 shows the Mannn-Whitney U values for the results. Run 5 appears to be marginally better than all of the other runs, but not significantly so. Otherwise all of the runs seemed to have performed more or less equally. This is probably due to the general inability to gain any good solutions to the problem.

Example Program Listing

```
# this program was not within the acceptable error range
# scoring 45sq pixels error
def fn0 ( input0 input1 )
( >
 ( gauss input1 )
 ( peak input0 )
)
```

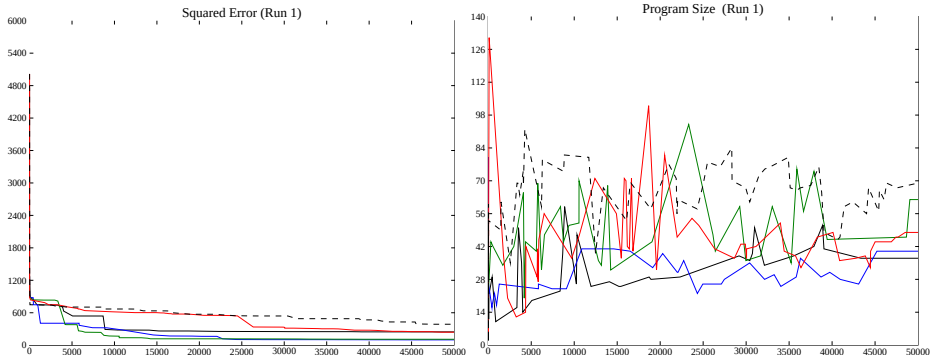

Figure 3.20: Moving Camera Run 1: Classic GP parameters.

```
def fn1 ( input0 )
( max
 ( +
  ( gauss ( * ( min 0 ( / input0 -1 ) ) ( + ( gauss input0 ) ( avg -1 ) ) ) )
  ( +
   ( gauss input0 )
   ( > ( avg ( max 0 -1) ) ( > ( / -1 1 ) ( diff 0 0 ) ) )
  )
 )
 ( +
  ( >
   ( cen input0 )
   ( peak ( diff ( neg -1 ) ( cen input0 ) ) )
  )
  ( +
   ( gauss ( gauss input0 ) )
   ( > ( avg ( max 0 -1 ) ) ( > ( / -1 1 ) ( diff 0 0 ) ) )
  )
 )
)
```

3.2.3 Reality Transfer

As with section 3.1.3, evolved programs from the static camera experiments (Section 3.2.1) were tried out using the test-harness program and webcam. The moving camera experiments were not tested, as they never yielded a program that performed within the desired error range.

As the data used for training relied on locating the bright ping-pong ball, the programs were tested by seeing how they reacted to a similarly bright object on a darker background. However they really reacted badly. The line used to

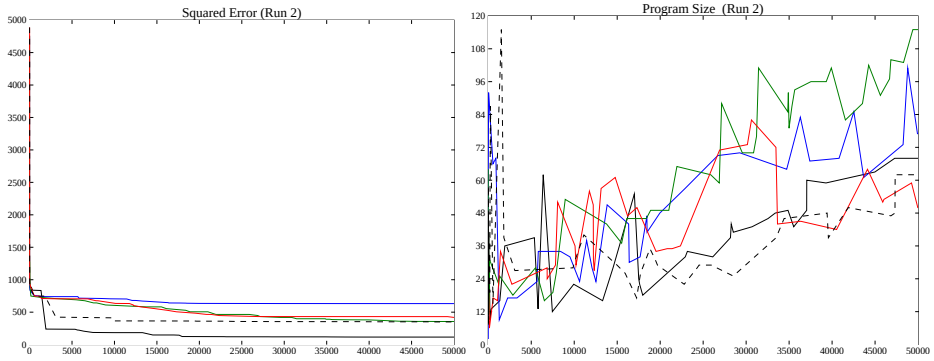

Figure 3.21: Moving Camera Run 2: Extra Mutation Operators.

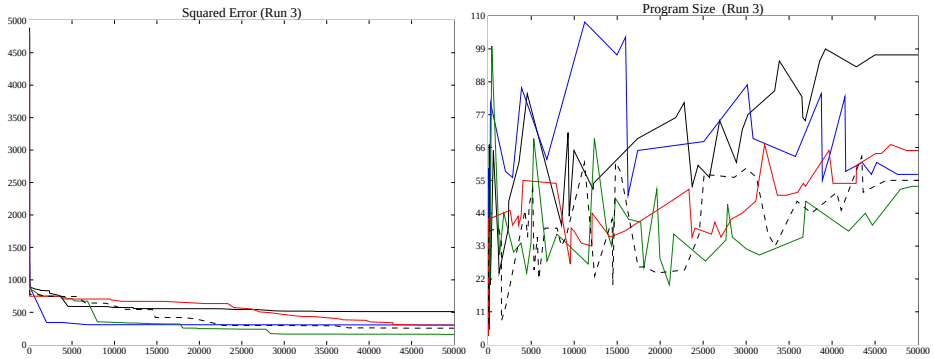

Figure 3.22: Moving Camera Run 3: Random Replacement.

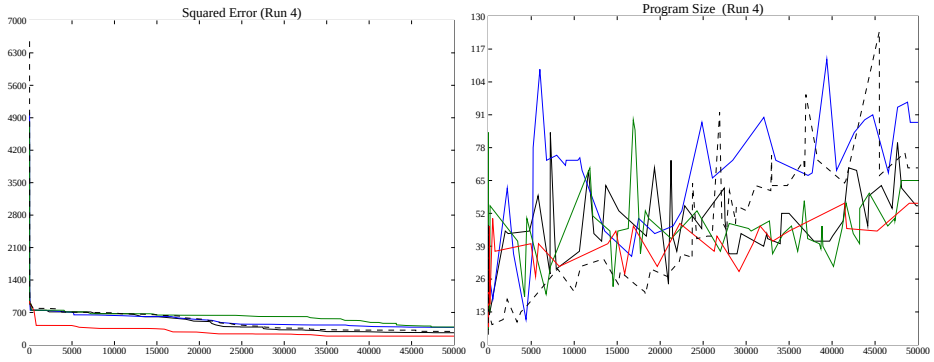

Figure 3.23: Moving Camera Run 4: Higher Mutation.

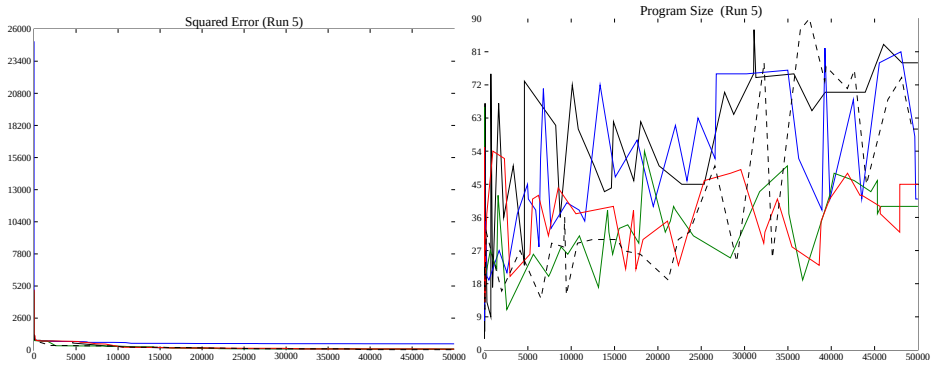

Figure 3.24: Moving Camera Run 5: Half Recombination - Half Mutation.

indicate position and velocity jumped around by very large amounts. This is hardly surprising considering the very small size and specialised nature of the evolved programs [7]. It seems that the programs were highly over-fitted to the data used for training, relying on the fact that the ping-pong ball contained the brightest pixels in the images and also that the ping-pong ball was fairly small in size. This meant that a very simplistic approach could usually be taken to "find" the ping-pong ball in training (hence the small program size), whereas when using the webcam this approach was too simplistic and the programs became distracted by noise and arbitrary bright spots in the field of view.

Perhaps to force the programs to generalise better it may have been a good idea to generate several versions of the testing data, by performing various simple operations on them. e.g.:

- Inverting the colours
- Altering Brightness
- Altering Contrast
- Rotating
- Mirroring

Those operations would help remove some of the peculiarities of the data. A major one of those being that the object of interest is always brightly coloured and in the top left hand corner.

[7] 6 nodes!

Chapter 4

Extensions, Improvements and Conclusions

4.1 Control Loop

Currently the controllers have only been evolved as "detectors" - they merely give us some information about the inputs they receive. The next logical step then would be evolve the controller on a task that requires both retrieving some information and then using that information to drive "actuators" in some fashion.

4.1.1 Tilt and Pan

The ultimate goal would be to drive motors to tilt and pan a camera, so that a moving object could be successfully tracked. This task requires some more complex behaviour, purely because the camera can now move. This means that the controller would have to compensate for this movement in some fashion, so that it knows when an object is actually moving, rather than the camera itself.

4.1.2 Pointing

An intermediate goal would be to create a simulated "pointing device", that would work on a static camera. The controller would be trained to move the pointing device around the cameras view, purely by controlling the on screen velocity of the pointer, with the aim of keeping the pointer on a moving object. This has the advantage of being a purely software solution and so being much easier to implement. It also removes the need to compensate for camera movement, but still presents an interesting control problem as the controller must both ascertain information about object position and/or velocity and then use that information to decide how to guide the pointer.

One would imagine that the function prototype required to perform this behaviour would look something like:

```
def fn0( input0, input1, pos, dt )
```

The first two inputs (input0 and input1) would correspond to successive camera frames as previously. The third input (pos) would be the pointers current on screen position and the fourth input (dt) would represent the time elapsed between the two input frames. The elapsed time would be required so that the objects true velocity can be calculated, rather than merely the instantaneous difference in positions between the two frames. It would also be necessary when creating the minimal simulation for training to vary dt sufficiently to make the use of that value more robust. Otherwise if it was always, say, 0.1 then the controller might come to depend on that exact value - perhaps even using dt in an inappropriate manner (e.g. in place of a constant).

Fitting in with the concept of "parallel shaping" we would probably also want to evolve the controller on the velocity and position detection tasks also. It would probably also be possible to train for these tasks using the current simulator, where object colour is important, but train the main controller on a simulator where object movement is the only useful indicator. This might sound a slightly perverse way of doing things, but the main controller would still be able to potentially use the other functions, by first calculating the (absolute) difference between the input frames - thus creating an input where colour is the key indicator.

4.2 More Operators

The OpenCV library is a really large library. I have only made use of a small subset of the computer vision operators available, so it would make sense to investigate adding some more functions for evolution to use. In particular some basic shape recognition operators would be very interesting. The ability to recognise simple circles, squares and lines might well be enough to allow the construction of more sophisticated recognition systems.

4.3 Mutation

In all of the four experiments performed Run 5 tended to produce the best results. This run made use of a higher mutation rate and a *lower* recombination rate. It was not always significantly better, in a statistical sense, than the other runs, but it did seem that mutation was a key factor in how well a run performed. Therefore more work needs to be done to investigate how much more mutation the system can handle and at what point it stops being useful. Perhaps a comparison should also be made between a genetic algorithm, with recombination, and simple hill-climbing/simulated annealing. It may also be the case that with more complex problems the apparent power of mutation may

not be quite as large. However it is very interesting that recombination does not seem to be the main mover of evolution, in this case, which is contrary Kozas [6] view.

4.4 Conclusion

I have successfully developed a C++, template based, genetic programming framework and image processing operators using IBMs OpenCV library. I have demonstrated this framework in action on several problems and have shown its ability to evolve solutions to some basic vision tasks. I have also demonstrated that program which perform well in Minimal Simulation can also happily make use of "real" sensory input, albeit for input that is not wildly dissimilar from that used in training. This is in constrast to the programs evolved on real data that were terribly over-fitted and under generalised and so failed to cross the "reality gap" [4].

There are still many more ways to test out and improve on the work I have done, but I feel it forms a solid basis to continue from.

References

[1] IBM. Open computer vision library. http://www.intel.com/research/mrl/research/opencv/.

[2] N. Jakobi. Evolutionary robotics and the radical envelope of noise hypothesis. *Adaptive Behavior*, 6:325–368, 1997.

[3] N. Jakobi. Evolving motion-tracking behaviour for a panning camera head. In J.A. Meyer R. Pfeifer, B. Blumberg and S. Wilson, editors, *From Animals to Animats 5*. MIT Press, 1998.

[4] N. Jakobi. The minimal simulation approach to evolutionary robotics. In T. Gomi, editor, *Evolutionary Robotics - From Intelligent Robots to Artificial Life (ER'98)*. AAI Books, 1998.

[5] I. Reid K. Bradshaw, P. McLauclan and D. Murray. Saccade and pursuit on an active head/eye platform. *Image and Vision Computing*, 12, April 1994.

[6] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[7] S. Meyer. *Effective C++: 50 Specific Ways to Improve Your Programs and Design*. Addison-Wesley, second edition, 1998.

[8] H. Montes and J. Wyatt. Graph representation for program evolution. Currently unpublished review paper. {h.a.montes, j.l.wyatt}@cs.bham.ac.uk, 2003.

[9] D. Murray, K. Bradshaw, P. McLauchlan, I. Reid, and P. Sharkey. Driving saccade to pursuit using image motion. *International Journal of Computer Vision*, 16:205–228, 1995.

[10] S. Perkins and Gillian Hayes. Evolving complex visual behaviours using genetic programming and shaping. *Interdisciplinary Approaches to Robot Learning*, June 2000.

[11] T.S. Ray. An approach to the synthesis of life. In J. Farmer C. Langton, C. Taylor and S. Rasmussen, editors, *Artificial Life 2*, Santa Fe Institue

Studies in the Sciences of Complexity, pages 371–408. Addison-Wesley, 1994.

[12] A. Robinson. Why visuomotor systems don't like negative feedback and how they avoid it. In *Vision, Brain and Cooperative Computation*, pages 89–107. MIT Press, 1987.

[13] A. Teller and D. Andre. PADO: A new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

[14] R.E. Keller W. Banzhaf, P. Nordin and F.D. Francone. *Genetic Programming an Introduction: On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

[15] R. Wilson. *Introduction to Graph Theory*. Longman, fourth edition, 1996.

Appendix A

Statement of Information Search Strategy

A.1 Forms of Literature

Conference papers are likely to be very important, as are journal articles. Books and technical articles will also be relevant, but mainly for specific information relevant to the implementation.

A.2 Appropriate Search Tools

The Science Citation Index will be used to find papers related to those already discovered from a review paper provided by my supervisor.

A.3 Search Statements

Searches on the Science Citation Index:

PERKINS S* 1998
JAKOBI N* MINIMAL SIMULATIONS

A.4 Search Evaluation

The above search terms yielded two records each.

Appendix B

Source Code

The C++ source code will be made available via the authors School of Computer Science personal website:

http://studentweb.cs.bham.ac.uk/∼msc37jxm/

Appendix C

OpenCV Functions.

- void cvAdd(const CvArr* A, const CvArr* B, CvArr* C); [1]
- void cvAddS(const CvArr* A, CvScalar S, CvArr* C); [1]
- void cvSub(const CvArr* A, const CvArr* B, CvArr* C); [1]
- void cvSubS(const CvArr* A, CvScalar S, CvArr* C); [1]
- void cvSubRS(const CvArr* A, CvScalar S, CvArr* C); [1]
- void cvMul(const CvArr* A, const CvArr* B, CvArr* C); [1]
- void cvDiv(const CvArr* A, const CvArr* B, CvArr* C); [1]
- void cvCmp(const CvArr* A, const CvArr* B, CvArr* C, int cmpOp);
- void cvCmpS(const CvArr* A, double S, CvArr* C, int cmpOp);
- void cvMax(const CvArr* A, const CvArr* B, CvArr* C);
- void cvMaxS(const CvArr* A, const CvArr* B, CvArr* C);
- void cvMin(const CvArr* A, const CvArr* B, CvArr* C);
- void cvMinS(const CvArr* A, const CvArr* B, CvArr* C);
- void cvAbsDiff(const CvArr* A, const CvArr* B, CvArr* C);
- void cvAbsDiffS(const CvArr* A, CvArr* C, CvScalar S);
- CvScalar cvAvg(const CvArr* A); [1]
- void cvMinMaxLoc(const CvArr* A, double* minVal, double* maxVal, CvPoint* minLoc, CvPoint* maxLoc); [1]
- void cvMoments(const CvArr* arr, CvMoments* moments); [1]

- double cvGetSpatialMoment(CvMoments* moments, int j, int i);
- void cvSmooth(const CvArr* src, CvArr* dst); [1]
- void cvErode(const CvArr* A, CvArr* C); [1]
- void cvDilate(const CvArr* A, CvArr* C); [1]
- void cvCanny(const CvArr* img, CvArr* edges, double threshold1, double threshold2); [1]

[1] Default parameters left out for clarity.