AN ERLANG IMPLEMENTATION OF DERIVING NATURAL LAWS THROUGH THE USE OF GENETIC PROGRAMMING

James Ashworth

APRIL 23, 2014

UNIVERSITY OF WARWICK
School of Engineering

Self-Assessment

The engineering contribution of this project is difficult to quantify at this stage. The program created is a tool, to be used initially by Dr Higgins's PhD group, but it is flexible enough to be used for any number of applications.

The field of Genetic Programming is currently growing in popularity, with more papers being published year on year for the last decade. The code itself is in a position to be extended by anyone with the time to learn the language, and the program can be run without any modifications to give the results presented in this report, and more.

In a year, this project has taken me from nothing, to learning a new language (Erlang), and a new paradigm (functional programming), to understanding Genetic Programming and its nuances, to writing a completely functional program using all of the above and proving, in testing, that it works exactly as intended. It is not without weaknesses, but these are mostly related to usability, rather than functionality, and have been addressed in the section 'Recommendation for Further Work'.

Summary

Project

This project was to create a program able to apply Genetic Programming to solve arbitrary data sets. The language had to be chosen, the code had to be written, and test cases had to be created and run. The program then had to be evaluated, to determine what modifications would need to be made before the program could be used by Dr Higgins's PhD Research Group.

Report

This report is structured to introduce the reader to the project, and then explain the basic tenets of Genetic Programming. Once the groundwork has been laid, the code itself is presented and documented, and taken forward to test. There are two types of testing performed for this report. Firstly, an in-depth look into how varying the parameters affects the outcome, and secondly, a broader look as the program tackles several separate data sets. The results of the testing are then discussed, and conclusions are drawn. Finally, the costs of the project are determined, and any shortcomings or new avenues to be explored are discussed. The appendices include the full table of results for the in-depth analysis of parameters, and the full commit log for the GitHub repository used to store and version the code.

Conclusion

Erlang proved to be very suitable for the project. The language is well documented by Eriksson, with a fast compiler, and a useful debugging function built in to the interpreter. At the conclusion, I have a good understanding of the complexities of Genetic Programming, especially after overcoming several problems during the coding process. The code itself now runs to completion without errors every time, and can solve fairly complex problems, given long enough. The project remains in a GitHub repository, giving a clear audit trail, and is fully commented, allowing it to be picked up as a future project if necessary. The report itself contains a very comprehensive result set, demonstrating the abilities (and limitations) of the project, across multiple data sets. The various requirements for future work are almost all cosmetic (in that they are not required for successfully running the program), which leads me to conclude that the project was a success, having met all of the goals stated in the original specification and exceeding many.

Table of Contents

Self-Assessment ii

Summary iii

Project iii

Report iii

Conclusion iv

Table of Contents v

List of Figures ix

List of Tables x

Introduction 1

Purpose 1

Conclusion 2

Demonstration of IT Skills 3

Final Project Specification 4

Language Decision 4

Understanding of GP 4

Code Writing 4

Presentation of Results 4

Literature Review / Theory 5

What is Genetic Programming? 5

How does Genetic Programming work? 7

Pool 7

Operations 7

Fitness 10

Methods 10

Overview of Code Structure 12

Language Decision 12

Representation of Equations 13

Representation of Elements within Equations 13

Evaluation 14

Of Equations 14

Of the Pool 14

Settings 15

Output 15

Equation Formatting 15

To File 15

Separation of Functionality 16

Initialise 16

Filereader 18

Operation 20

Evaluator 23

Output 25

Standard 27

Results 29

Roulette vs Stochastic Discussion 29

High Crossover vs Low Crossover 30

High Mutation vs Low Mutation 31

Multiple Mutation vs Single Mutation 32

No Duplicates vs Limited Duplicates vs Unlimited 33

Refilling Pool vs Not 34

Large Pool vs Small Pool 35

Approximations for $4\pi^2$ (39.478418, 6d.p.) 36

Circumference of a Circle 37

Convergence Times 37

Final Output 37

Volume of a Cuboid 38

Convergence Times 38

Final Output 38

Volume of a Square-Based Pyramid 39

Convergence Times 39

Final Output 40

Volume of a Torus 41

Convergence Times 41

Final Output 42

Results Analysis 43

Conclusions 44

Language Decision 44

Understanding of GP 44

Code Writing 44

Presentation of Results 45

Costing 45

Supervisor Time 45

Technician Time 45

Student Time 45

Printing Costs 45

Total Project Cost 45

Recommendation for Further Work 46

Code of Ethics and Professional Conduct 48

Bibliography 49

Appendix 1: Full Table of Results 52

Appendix 2: Commit Log and Statements 56

List of Figures

Figure 1 – Mutation of ((A + B) * C) to ((A / B) * C) 8

Figure 2 - Crossover of ((A + B) * C) and ((A - C) + (C * D)) 9

Figure 3 - Five 'spins' of roulette selection 10

Figure 4 - Two 'spins' of stochastic selection (each selecting five points) 11

Figure 5 - A generic contour graph showing a local maximum on the left and global maximum on the right 11

Figure 6 – ((A + B) * C) represented as a tree 13

Figure 7 - Bar chart showing the relative fitness of low and high crossovers, on the data set for the surface area of a torus 30

Figure 8 - Bar chart showing the relative fitness of low and high mutations, on the data set for the surface area of a torus 31

Figure 9 - Bar chart showing the relative fitness of single and multiple mutations, on the data set for the surface area of a torus 32

Figure 10 - Bar chart showing the relative fitness of 1, 3 and unlimited copies, on the data set for the surface area of a torus 33

Figure 11 - Bar chart showing the relative fitness of refilling the pool or not, on the data set for the surface area of a torus 34

Figure 12 - Bar chart showing the relative fitness of a small or large pool, on the data set for the surface area of a torus 35

Figure 13 - Graph showing the convergence times for the volume of a cuboid 38

Figure 14 - Graph showing the convergence times for the volume of a pyramid 39

Figure 15 - Graph showing the convergence times for the volume of a torus 41

List of Tables

Table 1 - Approximations for $4\pi^2$ as produced by the program 36

Table 2 - Equations and associated fitnesses for the circumference of a circle 37

Table 3 - Equations and associated fitnesses for the volume of a cuboid 38

Table 4 - Equations and associated fitnesses for the volume of a pyramid 40

Table 5 - Equations and associated fitnesses for the volume of a torus 42

Introduction

Purpose

This project was, at the highest level, to determine the feasibility of creating a program that used genetic programming to solve arbitrary problems. This can be divided into four distinct sections:

1. Initially, the language in which the program was to be written had to be decided. There were three languages suggested (MatLab, Python, and Erlang), which are discussed in detail under 'Language Decision' in the 'Overview of Code Structure' section.
2. Once chosen, the language had to be used to create the program, able to reliably cross and mutate equations to match a data-set. The code itself should be documented and logically laid out, with a focus on readability over efficiency. It should output the final results and at least rudimentary data showing performance. There should be the option to change most, if not all, of the rates and sizes through the use of parameters.
3. The program then needed to be tested, to verify that it could perform as required. Multiple data sets would be required, to show that the program can solve many problems. Ideally, an analysis showing how the program performs under different combinations of parameters, for a minimum of one data set as well. At this stage, basic geometry would be a reasonable area to begin.
4. Once the program was complete, it was to be evaluated to determine how it could benefit the research group's portfolio, and whether any further work would be necessary to match the group's requirements.

Conclusion

Having divided the purpose of the project into four sections, it seems reasonable to evaluate the conclusion against those same four sections.

1. The language chosen, Erlang, has proven itself to be more than adequate for a project of this type. The ability to generate multiple processes that can run independently and return to a controlling thread allows the program to make maximum use of the resources available.
2. The code written can be seen, in part, under 'Overview of Code Structure'. The majority of it has been documented in the report, and so in-line comments have been removed. There are 16 parameters that can be changed to tune the program according to requirements. The program outputs the final pool, with associated fitnesses, and the maximum fitness from each generation.
3. The program has been tested on 6 separate data sets, all of which can be seen under the heading 'Results'. The first section is a parameter sweep across a single data set, showing how the parameters interact with one another and what the effect is on the data set, along with a discussion for each parameter. The second section is a shorter trial against the remaining data sets, showing the solutions generated by the program, and the number of generations required to achieve them.
4. Currently, Dr Higgins is attempting to fit the outcome of this project to a problem that would be publishable in a peer-reviewed publication. Regardless, the program has some improvements to be made (under the heading 'Recommendation for Future Work'), and the code will be carried forward on to more complex problems, beyond the scope of a third year report.

Demonstration of IT Skills

This project has been written in Erlang, a parallel, functional programming language, using Notepad++, with XML to give correct syntax highlighting. It has then been run and debugged on the Erlang interpreter, supplied by Eriksson. The code itself has been uploaded to a GitHub repository, allowing versioning and coding on multiple machines. The parameters and results are stored in a Dropbox, allowing for multiple machines to run simulations to a single location.

This report has been written in Microsoft Word, with diagrams created using https://www.draw.io/, Microsoft Excel, Wolfram|Alpha, and Mathematica.

Final Project Specification

Language Decision

Three languages were put forward – MatLab, Python, and Erlang – each with their own benefits. One of these languages had to be chosen, although it was understood that, should the selected language prove untenable early enough in the process, the decision could be changed.

Understanding of GP

Dr Higgins was kind enough to suggest a couple of books and papers to read (Negnevitsky, 2005) (Bäck, Hammel, & Schwefel, 1997) (Nakano, Eckford, & Haraguchi, 2013). It was expected that I would learn enough about Genetic Programming methodology to replicate it in code, without going so far as to examine similar programs which may or may not be effective.

Code Writing

The bulk of this project was planned to be the code writing and debugging. The project was to be considered a success if any positive results could be achieved. The program was expected to be functional sometime around the middle of Term 2. The code itself needed to be written and commented clearly enough that it could be taken forward as part of another project.

Presentation of Results

There were two main results requirements – a broad test of the program's abilities, with multiple data sets, and an in-depth look at how changing the parameters of the program affected the results. These were to be included in the final report.

Literature Review / Theory

What is Genetic Programming?

Genetic Programming grew from the work of Nils Aall Barricelli, who started with evolutionary algorithms. His initial work covered genetic algorithms, the progenitor of GPs, whereby a bit-string representing a solution to a problem to be optimised is manipulated through cloning, crossover and mutation (Barricelli, 1957). Following on from this work, Lawrence J. Fogel applied GAs to finite-state automata (Fogel, Owens, & Walsh, 1966), which led to the first use of the tree structure in genetic programming, by Nichael L. Cramer (Cramer, 1985). This work was expanded on by John R. Koza, one of the main proponents of GP through the 1990s to now, who has written multiple books on the topic (Koza, 1992) (Koza, 1994) (Koza, Bennett, & Stiffelman, 1999) (Koza, et al., 2006). He is also the main benefactor of "The Humies" awards, discussed later.

Today, Genetic Programming is a field of computation currently best suited to areas where the general form of the solution is unknown (or the currently accepted form of the solution is thought to be wrong. It should be used when the general form of the solution is the goal, rather than an exact answer. It requires large amounts of test data, in computer readable format, and ideally there is a system in place to determine the fitness of any given solution (for example, in modelling – many simulators will be able to determine stresses and dynamics of an object without being able to offer improvements) (Poli, Langdon, & McPhee, 2008, pp. 111-113).

The form of Genetic Programming that this project maps to is Symbolic Regression – namely, determining a function that approximates an output given certain inputs. Existing examples of GPs being used for Symbolic Regression include:

- The creation of a soft sensor – a function to predict the conditions at a location by analysing the data at physical sensors nearby, where it would be difficult or expensive to place a real sensor (Jordaan, Kordon, Chiang, & Smits, 2004).
- Controlling the movement of a robot arm – using the data generated from the robot's "eyes" to control the actuators in the arm (Langdon & Nordin, 2001)
- Synthesis of analogue circuits – creating a schematic (and in some cases, routing maps) of amplifiers, mathematical operations (squares and square roots, cubes and cube roots, logarithms, etc.), thermometers and more (Koza, Andre, Bennet, & Keane, 1999).

Since 2004, an annual competition looking for Human-Competitive results of GP (results that either duplicate or improve on existing discoveries) called "The Humies" has been running and has had several impressive results[1]:

- A GP-designed antenna for use on a NASA satellite (Lohn, Hornby, & Linden, 2004).
- Automatic production of quantum computer programs (Spector, 2004).
- A system for detecting features in images under different transformations (Trujillo & Olague, 2006).
- A GP approach to finite algebras (Spector, Clark, Lindsay, Barr, & Klein, 2008).
- A GP approach to automated software repair (Forrest, Nguyen, Weimer, & Le Goues, 2009)

This is just a selection of the Gold Medal winners in the previous years – there have been significantly more, and it is worth noting that these are only the ones submitted for the award.

[1] http://www.genetic-programming.org/hc2011/combined.html (accessed 22/04/2014, 17:09)

How does Genetic Programming work?

Pool

The pool is the (ordered) collection of candidate equations. Each generation will use the old pool as a base, and will perform operations to generate a new pool.

Operations

Cloning

Cloning is the simplest of the operations – an equation is chosen from the current pool, and placed into the new pool.

Mutation

Mutation takes an equation from the current pool, and picks a random element in the equation (which could be a data-point, a constant, a variable or a function) and randomly swaps it for another element. This project only mutates an element to one of the same type. The mutated equation is then placed into the new pool.

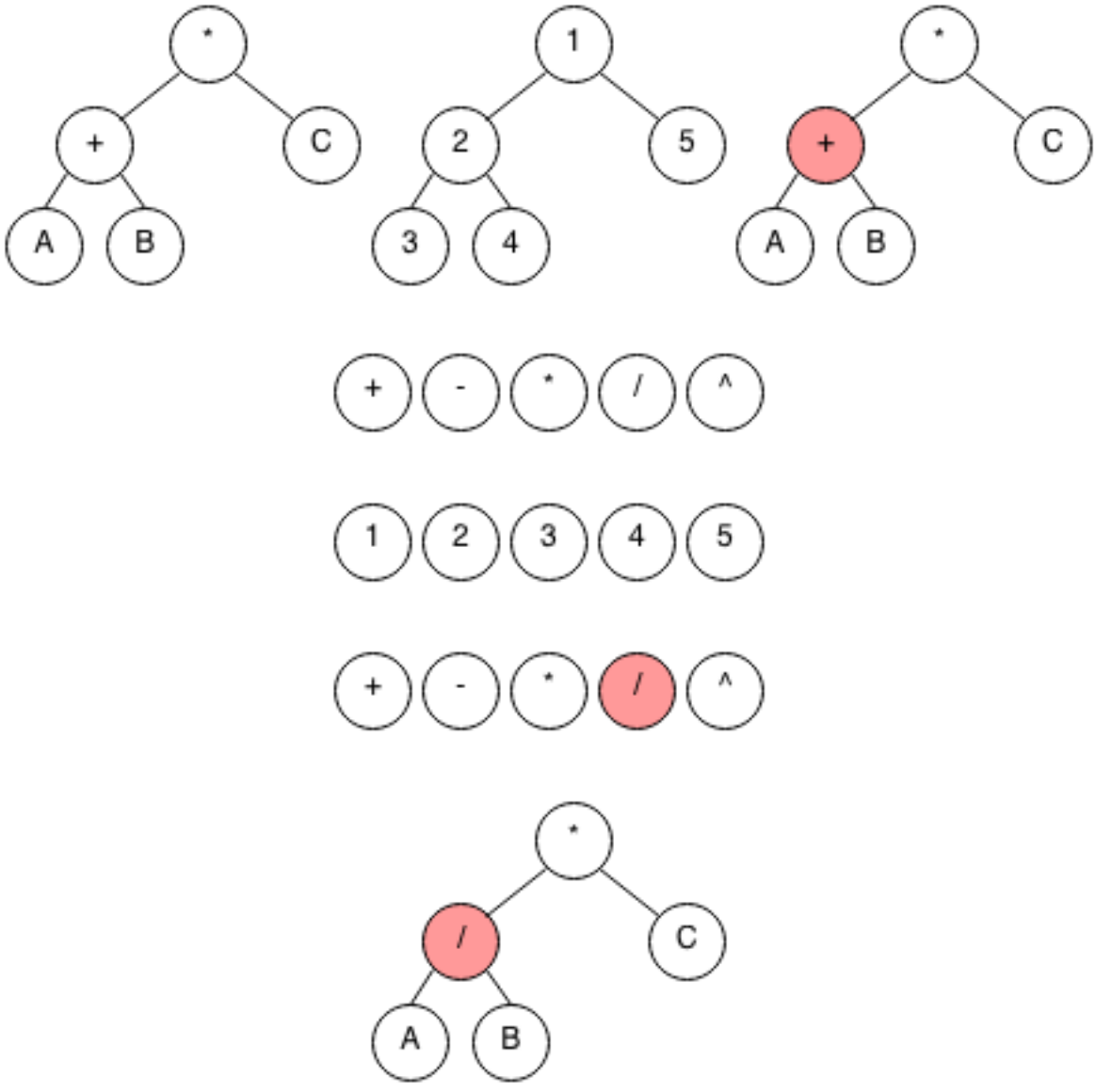

Figure 1 – Mutation of ((A + B) * C) to ((A / B) * C)

Crossover

Crossover takes two equations from the current pool (which could be identical), and picks random sections in each of them. These sections are then swapped between the two equations, and each equation is placed in the new pool.

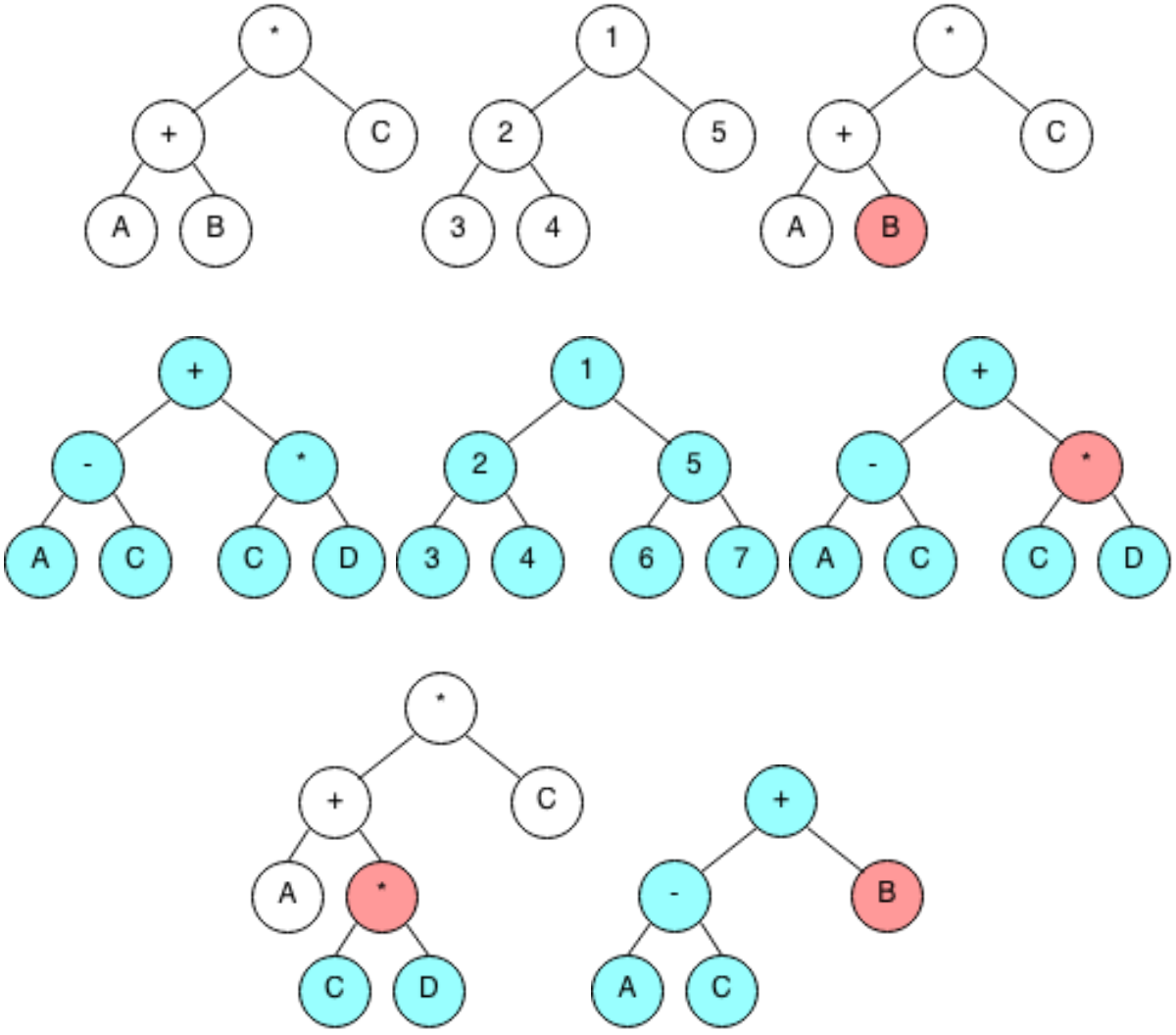

Figure 2 - Crossover of ((A + B) * C) and ((A - C) + (C * D))

Fitness

Once a new pool has been created, each item has to be evaluated so that the pool can be ordered. This measure is very loosely defined, and can change depending on the required result. The fitness function when looking for a short approximation will be different to the function to find an exact match for a complex system.

Methods

These methods are the system by which each operation can choose equations to manipulate.

Roulette

Roulette is a random process. Each selection is independent of previous selections. Better solutions are more likely to be chosen, but there is no guarantee that any given equation will be chosen. The standard analogy is that of a single pointer on a divided wheel – each section represents a candidate equation, with size equivalent to the fitness. Each selection is another spin of the wheel.

Figure 3 - Five 'spins' of roulette selection

Stochastic

Stochastic is a more tightly controlled process. The analogy is N pointers (where N is the number of candidates to be chosen in total), equally spaced over the same wheel as in Roulette. The pointers are then spun as a group, and each pointer is then used to determine a candidate equation. These equations are then placed in some sort of receptacle, from which

they are removed at random as required. This ensures that the best equation will be selected (as the size of that segment will be at worst $1/N^{th}$ of the wheel).

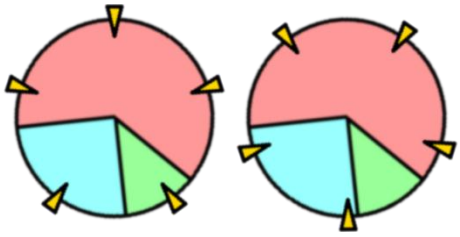

Figure 4 - Two 'spins' of stochastic selection (each selecting five points)

Tournament

Tournament is a method of pool combination, and can be used in addition to either Roulette or Stochastic. Without tournament, each previous pool is discarded in favour of the new pool. With tournament, the previous pool and new pool are combined, and the best N results are chosen to form the pool to be carried forward. This ensures that a particularly poor set of operations cannot destroy the pool, but can lead to the pool becoming trapped in a local maximum.

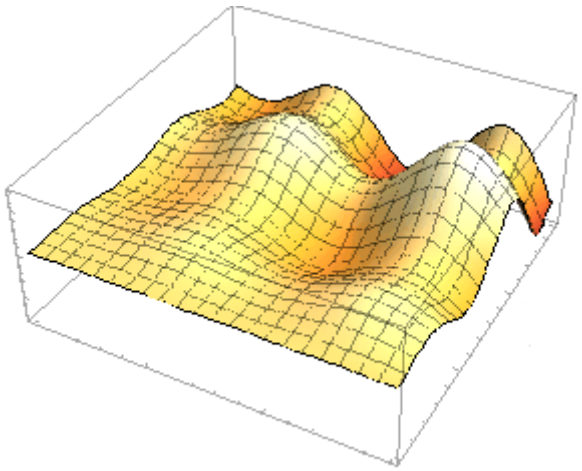

Figure 5 - A generic contour graph showing a local maximum on the left and global maximum on the right

Overview of Code Structure

Language Decision

Before the code could be written, the language needed to be chosen. There were three main candidates at the start of this project:

- MatLab is possibly the worst suited to a project of this type, with regards to language structure, but has libraries available for GPU support and is the best known by Dr Higgins.
- Python is significantly better suited, although still requires additional libraries to be suitable for the project, and is a language that I had some experience with.
- Erlang is a language that neither I nor Dr Higgins was aware of prior to the project, but has features that translate very well, such as built-in parallelism and a pure functional approach.

After some discussion, the decision was made to start with Erlang, with the option to re-evaluate after having had a chance to try the language.

Representation of Equations

In this program, equations are represented as a list of elements, in Polish notation. Each function (and associated arguments) is represented as a nested list within the equation. This could equally be considered as a tree structure.

$$A \times B \rightarrow [\times\ [A]\ [B]]$$

$$(A + B) \times C \rightarrow [\times\ [+[A]\ [B]]\ [C]]$$

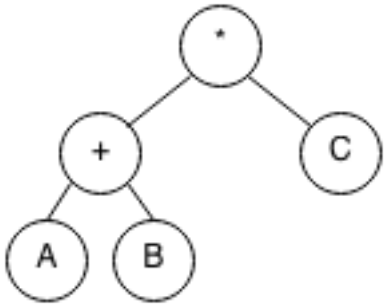

Figure 6 – ((A + B) * C) represented as a tree

Representation of Elements within Equations

Each element is represented by a tuple (a data structure), giving its type (as an atom), representation (as a string) and value (as a float, or a list of floats for each row of data). Functions are represented in much the same way, except that they have a lambda function defining the function that is to be performed on the operands and an arity (number of operands).

$$1 = \{\text{variable}, \text{"1.0"}, 1.0\}$$

$$\pi = \{\text{constant}, \text{"pi"}, 3.141\}$$

$$\text{radius} = \{\text{data}, \text{"radius"}, [1.0, 2.0, 3.0]\}$$

$$\times = \{\text{function}, \text{fun(X, Acc)} \rightarrow \text{Acc * X end}, \text{"*"}, 2\}$$

Evaluation

Of Equations

An equation is evaluated X times, where X is the number of data rows in the dataset. Each node is tested to determine whether it is a float. If the node is a function, each sub-node is tested. If a function has only floats as sub-nodes, that function is then replaced in the evaluation by the result of applying the function to the operands. If a node is a data-point, the list element for this evaluation, $x \leq X$, is taken as its value. In this way, the tree collapses down to a single result. If at any point the evaluation encounters an error (such as divide-by-zero, or a complex answer), the entire evaluation is returned as invalid. If the result returned is 0, the evaluation is returned as invalid. The result is then compared to the expected result for this evaluation, and the difference stored. Once all X runs are complete, the maximum difference and the spread of differences is returned. The difference is a good indicator of how close the equation is to correct, the spread is a good indicator of whether the data-points are present to the correct powers.

Of the Pool

Once each equation has returned with a maximum difference and a spread of differences, these values must be combined into a single fitness value. At the same time, the length of the equation is determined. On initialisation, a weighting is given to difference, spread and length, maximum values are given for difference and spread, and a range is given for length. Below this range, the length-fitness is 1, above this range the length-fitness is 0, and within the range is a linear progression. The difference and spread are then normalised between 1 and 0 (where 1 is best and 0 is worst). If any normalised fitness value is now 0, the overall fitness is

set to 0. If not, the three values are weighted, summed, and normalised to be between 1 and 0, and this is returned as the single fitness value for the equation.

Settings

The initial settings file reads in all of the required information: the location of the files containing the initial pool, the functions to be used, the constants available and the data-set; the pool-size and the levels of cloning, mutation (and the possibility of mutating again), and crossover; the maximum depth to which equations should be generated (to make up any disparity between the pool-size and the initial pool), whether the pool should be refilled between generations to replace equations that are invalid or have been removed as duplicates, and the maximum number of duplicates allowable for any equation (when set to the pool-size or higher, no duplicates will be removed – when set to 1, each equation in the pool must be unique); the cutoff values for difference, spread and length; and which files the results will be written to, and whether to overwrite any existing files or append.

Output

Equation Formatting

On being written to file, equations are converted from Polish notation to infix notation.

$$[\times\ [+[A][B]]\ [C]] \rightarrow ((A + B) \times C)$$

To File

Two files are written: the output file contains the pool as of the final generation, in descending order, with the fitness value for each equation; the tracking file contains the best fitness value from each generation, so that the progress of the program over the generations can be charted.

Separation of Functionality

Initialise

```
run_s(FileName) ->
    Date = date(),
    Time = time(),
    {A1, A2, A3} = now(),
    random:seed(A1, A2, A3),
    Settings = settings_file(FileName),
    io:fwrite("Settings Read",[]),
    erlang:spawn(initialise, cancel_monitor, [erlang:self()]),
    io:fwrite("~nGenerations Left:~n",[]),
    Results = operation:op_each_generation_s(settings_value(generations,
Settings),
                                    settings_value(numbers, Settings),
                                    settings_value(pool, Settings),
                                    settings_value(fcvsm, Settings),
                                    settings_value(fitnessVal, Settings),
                                    []),
    io:fwrite("~nWriting Logfiles~n",[]),
    output:output_to_file(settings_value(output, Settings), Date, Time,
Results).
```

This function is called to start the program. It logs the date and time the program was started, reseeds the RNG (Random Number Generator) using the current time (as otherwise the seed is the same for each run), reads in the full settings file, creates the cancel monitor process, begins outputting progress to the screen and sends off the main body of the program. Once the results are returned, it finishes by writing the logfiles.

```
cancel_monitor(PID) ->
    io:get_chars("(Press Enter to Cancel)", 0),
    erlang:send(PID, cancel).
```

This function runs in parallel with the main program – it monitors the main input window for the 'enter' key, and will send a cancel signal to the main program on receiving it.

```
settings_file(FileName) ->
    Params = filereader:read_settings_file(FileName),
    Constants = filereader:read_constant_file(key_value("constants",
                                                        Params)),
    Functions = filereader:read_function_file(key_value("functions",
                                                        Params)),
    Data = filereader:read_data_file(key_value("data", Params)),
    [Solution|Variables] = Data,
...
```

This is the first section of the function to read in the settings file – visible here is the initial reading of the parameters and each other file specified in parameters.

```
generate_pool(Pool, Size, FCVSM) when is_list(Pool) ->
    Pool ++ generate_pool(length(Pool), Size, FCVSM);
generate_pool(Pool, Size, {Fs, Cs, Vs, Solution, _MultiMutation, MaxDepth})
                                                        when Pool < Size ->
    NewEquation = create_equation(Fs, Cs, Vs, MaxDepth),
    NewElement = {NewEquation, evaluator:evaluate(Solution, NewEquation)},
    if
        element(1, element(2, NewElement)) == invalid ->
            generate_pool(Pool, Size, {Fs, Cs, Vs, Solution, 0, MaxDepth});
        true ->
            [NewElement] ++ generate_pool(Pool+1, Size, {Fs, Cs, Vs,
                                                    Solution, 0, MaxDepth})
    end;
generate_pool(_Pool, _Size, _FCVSM) ->
    [].
```

This code increases the pool to the required size – the initial implementation had poor memory usage due to the invalid equations being held on to. The new code (shown above) is tail-recursive, so is more memory efficient.

```
do_create_equation(Fs, Cs, Vs, MaxDepth, FProb) ->
    if
        MaxDepth > 0 ->
            FChoice = random:uniform();
        true ->
            FChoice = 1.0
    End,
...
```

This is the initial section of the equation creation code. A function will be chosen if the value of FChoice is lower than the probability of a function, FProb (assigned recursively). If the equation is now at the maximum depth, FChoice is set to 1.0 – this will not be lower than FProb, so a constant, data, or variable will be inserted. If not, a random FChoice between 0 and 1 is generated. If FChoice is lower, a function will be inserted at random and its arity inspected. Depending on the arity, do_create_equation will be called recursively either once or twice more, to create the operands.

Filereader

```
read_data_file(FileName) ->
    Return = file:read_file(FileName),
    if
        element(1, Return) == ok ->
            Binary = binary:bin_to_list(element(2, Return)),
            Rows = string:tokens(Binary, "\r\n"),
            Data = lists:map(fun(X) -> string:tokens(X, ",") end, Rows),
            DataT = data_transpose(Data),
            ToFloatFunc = fun(Z, Acc) ->
                              lists:append(Acc,
                                           [standard:std_string_to_float(Z)]) end,
            ToTuplesFunc = fun(Y) ->
                              [Hy|Ty] = Y,
                              {data, Hy, lists:foldl(ToFloatFunc, [], Ty)}
                              end,
            lists:map(ToTuplesFunc, DataT);
        true ->
            error
    end.
```

This is an example of the code used to read in a file – specialised versions exist for each type of file, although the basic principles demonstrated here are fairly unchanged. The file is read in, in binary format. The binary is then turned into a string (a string in Erlang being a list of characters), and broken by line break characters (using the Windows standard line break). In the case of the data file, each row is then broken into elements by comma-separation. The data now exists as a series of data rows, where each row is a record, including a row for headers. As such, the data must be transposed, to create a series of column records, where each column is a data-variable. The data must then be collated into tuples, which is done as a batch by defining a lambda function which is then mapped across the entire data-set.

```
assemble_equation(Equation, Distance) when length(Equation) > 0 ->
    [H|T] = Equation,
    if
        Distance > 0 ->
            if
                element(1, H) == function ->
                    assemble_equation(T, Distance + element(4, H) - 1);
                true ->
                    assemble_equation(T, Distance - 1)
            end;
        true ->
            if
                element(1, H) == function ->
                    if
                        element(4, H) == 2 ->
                            [H, assemble_equation(T, 0),
                                                assemble_equation(T, 1)];
                        true ->
                            [H, assemble_equation(T, 0)]
                    end;
                true ->
                    [H]
            end
    end;
assemble_equation(_Equation, _Distance) ->
    error.
```

This code assembles a flat list of Polish notation elements into the nested list format used elsewhere. Initially called with Distance = 0, it finds the first element. If the element is not a function, it is returned as a sub-list. If the element is a function, it must be returned with its arguments. For a function of arity 1, this is the next element, so it recurs with the remainder of the flat list and Distance = 0. For a function of arity 2, however, it needs to be aware of the possibility of a function as the first argument. It therefore recurs twice, once with Distance = 0, once with Distance = 1. When Distance > 0, if the next element is a function, its arity is used to increase the count for recursion, so it can skip over any arguments to the sub-function. Thus, the entire flat-list can be deconstructed and reconstructed as a nested list of expected format. If the flat-list is longer than expected, it will be truncated. If it is shorter, the equation will return with an error, and will be filtered out on evaluation.

Operation

```
op_each_generation_s(GensLeft, Numbers, Pool, FCVSM, Standard, Record) when
                                                        GensLeft > 0 ->
    output:write_gens_left(GensLeft),
    Total = lists:foldl(fun(X, Acc) -> Acc + element(2, X) end, 0, Pool),
    {CloneNo, CrossNo, MutateNo} = Numbers,
    Size = CloneNo + CrossNo + MutateNo,
    Offset = (Total / Size) * random:uniform(),
    Pointers = op_make_pointers(Size, Offset, Total / Size),
    [H|T] = op_each_operation_s(Numbers, Pool, FCVSM, Pointers),
    BestEq = element(1, lists:last(Pool)),
    Solution = element(4, FCVSM),
    Hold = [{BestEq, evaluator:evaluate(Solution, BestEq)}],
    if
        H == cancelled ->
            NewPool = standardise_pool(T ++ Hold, Standard, FCVSM),
            op_each_generation_s(0, Numbers, NewPool, FCVSM, Standard,
                                 Record ++ [element(2, lists:last(NewPool))]);
        true ->
            NewPool = standardise_pool([H] ++ T ++ Hold, Standard, FCVSM),
            op_each_generation_s((GensLeft - 1), Numbers, NewPool, FCVSM,
                     Standard, Record ++ [element(2, lists:last(NewPool))])
    end;
op_each_generation_s(_GensLeft, _Numbers, Pool, _FCVSM, _Standard, Record)
                                                                    ->
    {Pool, Record}.
```

This is the stochastic version of the code run for each generation. The roulette version is similar, but with some omissions because of the less complicated selection method. For each generation, it writes to the screen to update progress. The total of the fitness values in the pool is determine, and divided by the required number of pointers to get the separation between pointers. The new pool is then generated through the operations. Once the new pool is retrieved, a slight tournament bypass is utilised to maintain the best equation from the previous pool (so that the best solution so far cannot be lost) and check for the cancel signal. If it has been cancelled, this pool is standardised and it recurs with no remaining generations. If not, it standardises and recurs with one fewer generation remaining.

```
op_each_operation_s({CloneNo, CrossNo, MutateNo}, Pool, FCVSM, Pointers)
                                                        when CrossNo > 0 ->
    Pointer1 = lists:nth(random:uniform(length(Pointers)), Pointers),
    PointersI = lists:delete(Pointer1, Pointers),
    Pointer2 = lists:nth(random:uniform(length(PointersI)), PointersI),
    PointersN = lists:delete(Pointer2, PointersI),
    _PID = erlang:spawn(operation, op_new_generation, [Pool, crossover,
                                FCVSM, erlang:self(), Pointer1, Pointer2]),
    NewPool = op_each_operation_s({CloneNo, (CrossNo - 2), MutateNo}, Pool,
                                                          FCVSM, PointersN),
    op_receiver(NewPool);
...
```

This is the first section of the operations code for stochastic. As before, roulette is similar but does not require pointers, so will not be shown. This deals with crossover – mutation and cloning follow. For each crossover, two pointers are removed at random from the list of pointers, and a parallel sub-process is started to perform the crossover. It then recurs, until all of the processes (of all types) have been started. As each sub-process finishes, it sends a message back to the main process with its results. These messages are stacked until the main process is ready to receive them, which it does on the way back up.

```
op_do_new_generation(Pool, Type, {_Fs, _Cs, _Vs, Solution, _MultiMutation,
                                  _MaxDepth}, R1, _R2) when Type == clone ->
    Return = element(1, op_do_get_equation(Pool, R1)),
    {Return, evaluator:evaluate(Solution, Return)};
op_do_new_generation(Pool, Type, {Fs, Cs, Vs, Solution, MultiMutation,
                                 _MaxDepth}, R1, R2) when Type == mutation ->
    Equation = op_do_get_equation(Pool, R1),
    A1 = trunc(R2 * 100),
    A2 = trunc(R2 * 10000) rem 100,
    A3 = trunc(R2 * 1000000) rem 10000,
    random:seed(A1, A2, A3),
    Return = op_mutation(element(1, Equation), Fs, Cs, Vs, MultiMutation,
random:uniform()),
    {Return, evaluator:evaluate(Solution, Return)};
...
```

This is the main body of the parallel sub-process that deals with cloning and mutation (crossover follows). Cloning is very simple. An equation is selected, evaluated and returned. Mutation, by comparison, is more complex because of the need to reseed the RNG for each sub-process (otherwise each sub-process would have the same 'random' numbers). A random number from the parent process is used to reseed – the first six decimal places are used to

generate three two-digit numbers for the seed. The selected equation is then mutated, evaluated, and returned.

```
op_do_mutation(X, R1, Fs, Cs, Vs) when (R1 > 1) ->
    [Hx|Tx] = X,
    N = list_length(Hx),
    if
        N < R1 ->
            [Hx] ++ op_do_mutation(Tx, R1 - N, Fs, Cs, Vs);
        true ->
            [op_do_mutation(Hx, R1, Fs, Cs, Vs)] ++ Tx
    end;
op_do_mutation(X, _R1, Fs, Cs, Vs) ->
    [Hx|Tx] = X,
    if
        is_list(Hx) ->
            [op_do_mutation(Hx, 1, Fs, Cs, Vs)] ++ Tx;
        true ->
            if
                element(1, Hx) == function ->
                    op_do_function_mutation(Hx, Tx, Fs);
                element(1, Hx) == constant ->
                    [lists:nth(random:uniform(lists:flatlength(Cs)), Cs)];
                element(1, Hx) == data ->
                    [lists:nth(random:uniform(lists:flatlength(Vs)), Vs)];
                element(1, Hx) == variable ->
                    [op_do_variable_mutation(Hx)];
                true ->
                    error
            end
    end.
```

This finds the location in the equation to be mutated in the first instance, then determines the type of the element at that location. This element is then replaced with a random element of the same type (nb: there is no guard against replacing it with the same element). In the instance of variable mutation, the variable will be incremented, decremented, or left unchanged.

```
op_do_crossover(X, Y, R1, R2) when (R1 > 1) ->
    [Hx|Tx] = X,
    N = list_length(Hx),
    if
        N < R1 ->
            [Hx] ++ op_do_crossover(Tx, Y, R1 - N, R2);
        true ->
            [op_do_crossover(Hx, Y, R1, R2)] ++ Tx
    end;
op_do_crossover(X, Y, R1, R2) when (R2 > 1) ->
    [Hy|Ty] = Y,
    N = list_length(Hy),
    if
        N < R2 ->
            op_do_crossover(X, Ty, R1, R2 - N);
        true ->
            op_do_crossover(X, Hy, R1, R2)
    end;
op_do_crossover(X, Y, _R1, _R2) ->
    [Hx|Tx] = X,
    [Hy|_Ty] = Y,
    if
        is_list(Hx) ->
            [op_do_crossover(Hx, Y, 1, 1)] ++ Tx;
        true ->
            if
                is_list(Hy) ->
                    op_do_crossover(X, Hy, 1, 1);
                true ->
                    Y
            end
    end.
```

This code is called with two equations and two random points. The first equation is disassembled recursively until the first point is identified, then the second equation is disassembled until the second point is identified. The first equation is then reassembled with the second point embedded.

Evaluator

```
evaluate(Solution, Algorithm) ->
    No = lists:foldl(fun(_, Acc) -> Acc + 1 end, 0, element(3, Solution)),
    Fitnesses = do_evaluate(Solution, Algorithm, No),
    Check = lists:max(Fitnesses),
    if
        Check == invalid ->
            {invalid, 0, 0};
        true ->
            {lists:max(Fitnesses), lists:max(Fitnesses) -
                    lists:min(Fitnesses), length(lists:flatten(Algorithm))}
    end.
```

First the number of data rows is determined by using a basic accumulator across the solution list. Then the equation is evaluated against these solutions, returning a list of values. This list

is checked for invalid responses, and, if any are found, returns an invalid tuple. Otherwise, the worst value, the spread of values, and the length of the equation are returned in a tuple.

```
evaluate_fitness(Solution, Algorithm, No) ->
    Temp = lists:keyfind(data, 1, lists:flatten(Algorithm)),
    if
        Temp == false ->
            invalid;
        true ->
            Value = (catch do_compute(Algorithm, No)),
            if
                element(1, Value) == 'EXIT' ->
                    invalid;
                Value == 0 ->
                    invalid;
                true ->
                    ExValue = lists:nth(No, element(3, Solution)),
                    abs((Value - ExValue) / ExValue)
            end
    end.
```

Any form of data-point is searched for in the equation being checked. If not found, it is returned as invalid immediately. A try-catch block is then used to compute the value of this iteration – in the event of a divide-by-zero, a complex result, or some other unexpected error the code returns an 'EXIT' tuple, and recovers (rather than crashing the process). If the value is 0, the equation is marked as invalid (0 results are a problem for the fitness, being exactly one expected result from the expected result at all times). Otherwise, it returns the magnitude of how far from the expected value this iteration is, with regards to the expected value.

```
do_compute([H|T], No) ->
    N = length(T),
    if
        N == 0 ->
            if
                element(1, H) == constant ->
                    element(3, H);
                element(1, H) == variable ->
                    element(3, H);
                element(1, H) == data ->
                    lists:nth(No, element(3, H));
                true ->
                    false
            end;
        N == 1 ->
            [T1|_] = T,
            lists:foldl(element(2, H), 0, [do_compute(T1, No)]);
        N == 2 ->
            [T1,T2|_] = T,
            lists:foldl(element(2, H), do_compute(T1, No), [do_compute(T2,
                                                                       No)]);
        true ->
            false
    end.
```

This works recursively down the list, computing the value at each sub-node from the bottom up. If the node is a function, we fold the lambda function over the following operands, once they have been evaluated.

Output

```
printable([H|T], Brackets) ->
    N = length(T),
    if
        N == 0 ->
            if
                Brackets == true ->
                    "(" ++ element(2, H) ++ ")";
                true ->
                    element(2, H)
            end;
        N == 1 ->
            [T1|_] = T,
            element(3, H) ++ printable(T1, true);
        N == 2 ->
            [T1,T2|_] = T,
            "(" ++ printable(T1, false) ++ " " ++ element(3, H) ++ " " ++
printable(T2, false) ++ ")";
        true ->
            false
    end.
```

Another recursive function, this time to rearrange from Polish to infix notation. Any single element is returned as is, either with or without brackets, depending on its parent function.

A function with a single argument is appended to its argument, with brackets around the argument (ie $[\sin [X]]$ becomes $\sin(X)$). A function with two arguments is placed between its arguments, without brackets on the arguments but with brackets around the entire function (ie $[\times [A] [B]]$ becomes $(A \times B)$).

```
output_to_file({PoolFileName, TrackingFileName, OutMode}, Date, StartTime,
                                                        {Pool, Record}) ->
    if
        OutMode == "overwrite" ->
            PoolFile = element(2, file:open(PoolFileName, [write])),
            TrackingFile = element(2, file:open(TrackingFileName,
                                                                [write]));
        true ->
            PoolFile = element(2, file:open(PoolFileName, [append])),
            TrackingFile = element(2, file:open(TrackingFileName,
                                                                [append]))
    end,
    write_line([PoolFile, TrackingFile],
                list_to_binary(integer_to_list(element(3, Date)) ++ "/" ++
                                    integer_to_list(element(2, Date)) ++ "/"
                                     ++ integer_to_list(element(1, Date)))),
    write_line([PoolFile, TrackingFile],
             list_to_binary(integer_to_list(element(1, StartTime)) ++ ":" ++
                                integer_to_list(element(2, StartTime)) ++ ":"
                                 ++ integer_to_list(element(3, StartTime)))),
...
```

Upon starting the output, the files must be opened. If the parameters specified overwriting the files, they are opened in ‘write’ mode, which will erase the existing file and create a new one. If anything other than “overwrite” is specified, the program appends for safety. The files are initialised with the date and time that the program was started, and further down the specialised functions export the data and tracking information to each file. Finally, the time at which the process finished is printed, and the files are closed.

```
output_to_pool_file(OutputFile, Pool, Last) when (length(Pool) > 0) ->
    [H|T] = Pool,
    if
        H == Last ->
            output_to_pool_file(OutputFile, T, H);
        true ->
            output_to_pool_file(OutputFile, T, H),
            if
                element(2, H) > 0 ->
                    write_partial_line([OutputFile],
                                       float_to_binary(element(2, H)
                                           * 1.0, [{decimals, 6}])),
                    write_partial_line([OutputFile], <<" - ">>),
                    write_line([OutputFile],
                               list_to_binary(printable(element(1, H))));
                true ->
                    ok
            end
    end;
output_to_pool_file(_OutputFile, _Record, _Last) ->
    ok.
```

The tracking file is straightforward in that each value is output sequentially, whereas the pool file has some additional post-processing. The records are stored in reverse order by fitness, so the list is traversed and output is done from last to first on the way back up. Each record is checked to ensure that it is different to the record before (records are sorted by fitness and then by 'alphabet', so any identical records will be coincident) and that it has a fitness value greater than zero (which may be the case for automatically generated records if the pool is being refilled, as well as casualties of the final set of operations). If it passes both of these, it is output with its fitness to file.

Standard

```
std_float_to_list(Value) ->
    float_to_list(Value, [{decimals, 10}, compact]).
```

This is a helper function to allow for changes across the program in terms of formatting. Here, all floats are changed to have a maximum of 10 decimal places, with a minimum of one.

```
std_string_to_float(Value) ->
    Index = string:str(Value, "."),
    if
        Index > 0 ->
            element(1, string:to_float(Value));
        true ->
            element(1, string:to_float(string:concat(Value, ".0")))
    end.
```

This is a helper function to read in integers. Erlang's string-to-float conversion gives an error for an integer with no decimal point, so we look for a decimal point, and append ".0" if one cannot be found before conversion.

Results

For the discussion sections, the same data-set (surface area of a torus) was run 288 times, with every permutation of variables being run three times to reduce variance. Each output was then ranked 1, 2, 3 or X. A rank of 1 was awarded for each run that produced the 'perfect' solution ($4 \times \pi \times \pi \times r \times R$) or variants thereof ($4 \times \pi^2 \times r \times R$ and $(2 \times \pi)^2 \times r \times R$, as well as permutations on these), a rank of 2 was awarded for a 'perfect' solution with additional 'do nothing' sections (N^1, $N \times 1$, $N + 0$, etc.), and a rank of 3 was awarded for any solution that approximated $4\pi^2$ to within 1% accuracy. All other solutions were ranked 'X', and are considered to be failures (within the generation-limit imposed). All trials were given 10^6 operations – therefore the 'large' pool of 1,000 equations was given 1,000 generations, and the 'small' pool of 100 equations was given 10,000 generations.

Roulette vs Stochastic Discussion

The decision was made to use Stochastic after discussion with my supervisor, due to its inability to accidentally regress (Roulette being able to exclude the best solution, the chance of which increases with the number of generations).

High Crossover vs Low Crossover

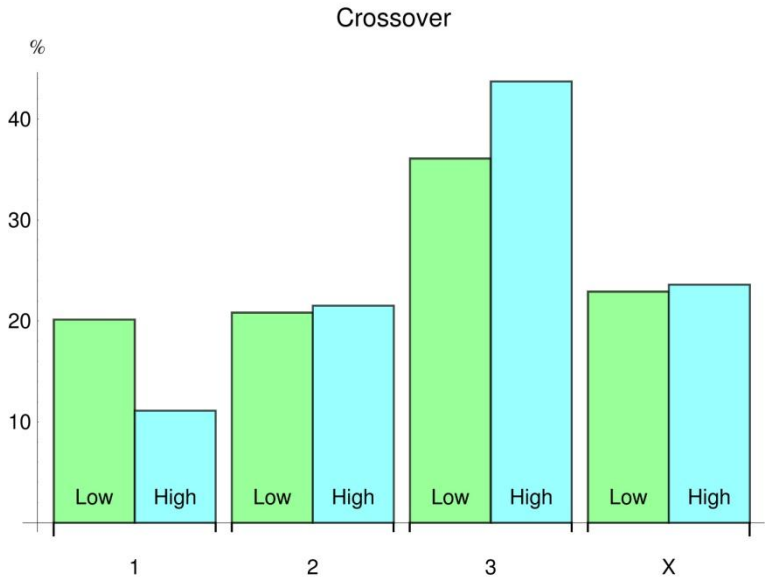

Figure 7 - Bar chart showing the relative fitness of low and high crossovers, on the data set for the surface area of a torus

From the results obtained we can see that low and high crossover both give a similar failure rate, but low crossover is better for finding an exact solution (where one exists), and high crossover is better for finding an approximation.

High Mutation vs Low Mutation

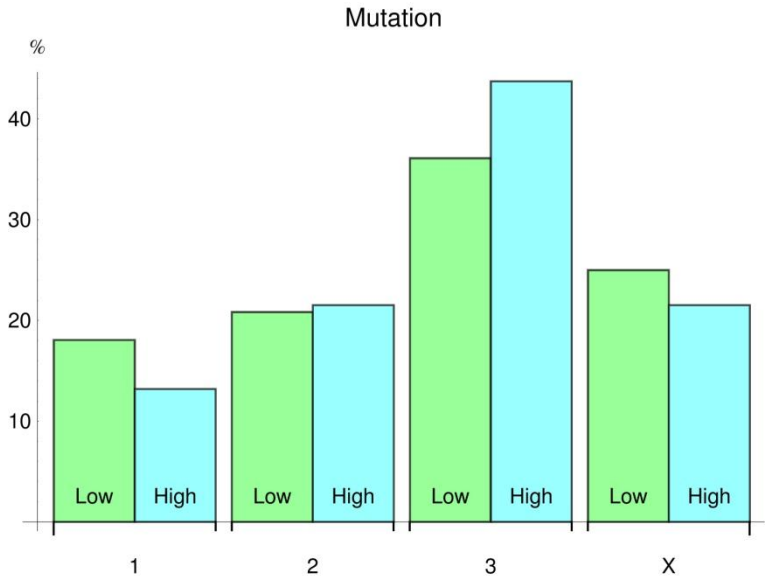

Figure 8 - Bar chart showing the relative fitness of low and high mutations, on the data set for the surface area of a torus

This chart shows that, while a low rate of mutation is more likely to give an unusable result, it also has a higher chance of finding the best solution. As with crossover, higher mutation is better for generating an approximate solution.

Multiple Mutation vs Single Mutation

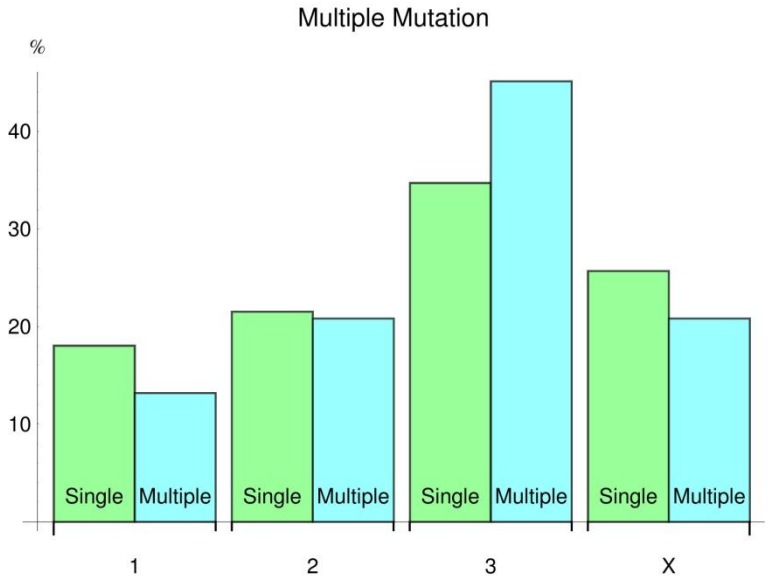

Figure 9 - Bar chart showing the relative fitness of single and multiple mutations, on the data set for the surface area of a torus

When the possibility of mutating a solution a second time (or more) is introduced, we reduce the probability of finding the optimum solution but increase the probability of having a usable result, and giving a good approximation.

No Duplicates vs Limited Duplicates vs Unlimited

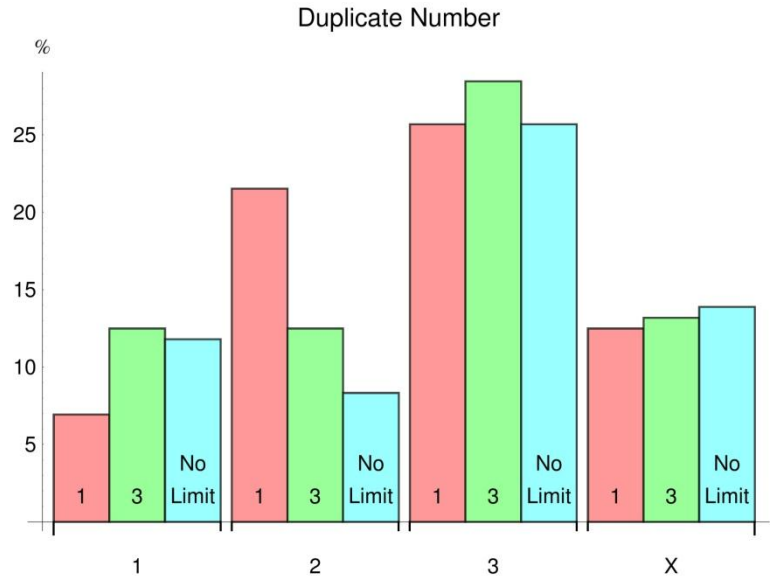

Figure 10 - Bar chart showing the relative fitness of 1, 3 and unlimited copies, on the data set for the surface area of a torus

An intermediate number of duplicates gives the highest chance of producing the correct answer and giving a good approximation, but is significantly less likely to give a solution that merely requires simplification to be considered optimum than a pool composed of unique solutions. Removing the limit introduces the possibility of monotonicity, and does not have a clear advantage in any area. It is worth noting that the number of duplicates required for any data set will vary depending on the pool size chosen.

Refilling Pool vs Not

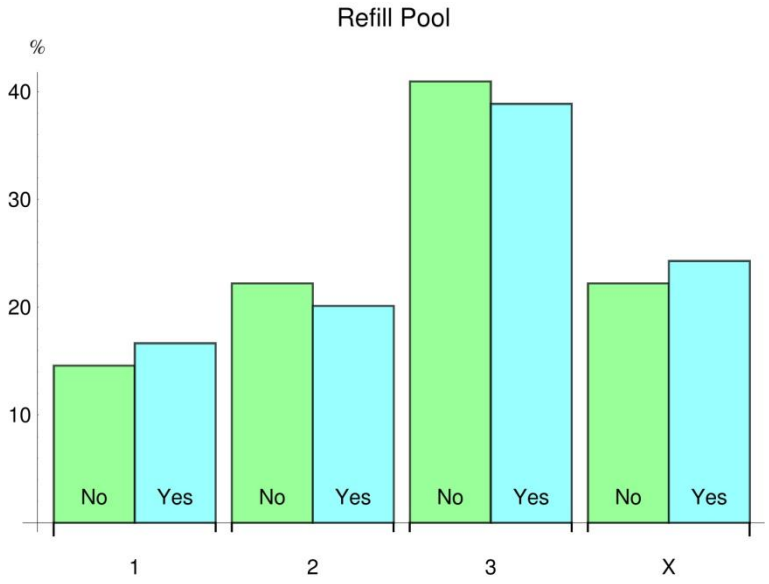

Figure 11 - Bar chart showing the relative fitness of refilling the pool or not, on the data set for the surface area of a torus

The results set suggests that whether the pool is refilled or not makes very little difference to the quality of the results. In most instances, the generated equations will be of low fitness in comparison with the existing pool, apparently to the point that they are, for all intents and purposes, a waste of processing power.

Large Pool vs Small Pool

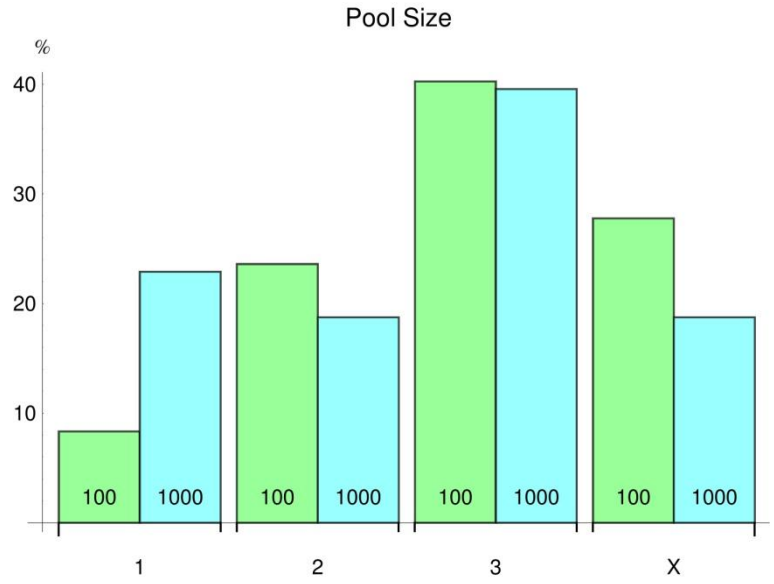

Figure 12 - Bar chart showing the relative fitness of a small or large pool, on the data set for the surface area of a torus

In comparison with the previous analyses, the question of pool size has a very definite answer. A small pool is much more likely to produce unusable results, and less likely to harbour a correct result. As such, the pool size should be the maximum that can be handled by the hardware available.

Approximations for $4\pi^2$ (39.478418, 6d.p.)

Equation	Representation	Value	Offset (%)
1	$\frac{377\pi}{30}$	39.479348	0.00235591
2	$\frac{65e+455}{16}$	39.480520	0.00532525
3	$3e\pi+3\varphi+9$	39.473305	0.01295130
4	$5\varphi^2+5e+15$	39.483697	0.01337282
5	$4\varphi+33$	39.472136	0.01591160
6	6×2^e	39.485316	0.01747370
7	$e(5^\varphi+1)$	39.467805	0.02688256
8	$9e+15$	39.464536	0.03516136
9	$\frac{88\pi}{7}$	39.494308	0.04024994
10	$\pi^\pi+3$	39.462160	0.04118199
11	$\frac{6e\pi}{\varphi}+\frac{12e\pi}{5\varphi^2}$	39.495608	0.04354315
12	39.5	39.500000	0.05466885

Table 1 - Approximations for $4\pi^2$ as produced by the program

Many of these approximations are random, and it is coincidental that they are a good fit for $4\pi^2$. Equation [9], however, is equivalent to the original Ancient Egyptian approximation for π, of $\frac{22}{7}$, and equation [1] is close to the 4th approximation of the continued fractional representation of π, $\frac{355}{113}$[2].

[2] http://en.wikipedia.org/wiki/Approximations_of_%CF%80 (accessed 14/04/14, 16:17)

Circumference of a Circle

Convergence Times

In the second and third runs, the best solution was one of the originally generated solutions.

In the first run, it took a single generation to arrive at the optimum solution.

Final Output

Position	1st Run		2nd Run		3rd Run	
	Equation	Fitness	Equation	Fitness	Equation	Fitness
1	$\pi(r+r)$	1.00000	$2\pi r$	1.00000	$2r\pi$	1.00000
2	$r(\pi+\pi+r^{-7^{r^r}})$	1.00000	$r(5+e^{\frac{2^1}{8}})$	0.99987	$2r(\pi-\frac{r}{r^{6^{\varphi}}})$	0.99999
3	$r(\pi+\pi+r^{-7^{r\times r}})$	1.00000	$r(5+e^{\frac{2^0}{4}})$	0.99987	$2r(\pi-\frac{5}{r^{4^e}})$	0.99999
4	$r(\pi+\pi+r^{-7^{r+r}})$	1.00000	$r(5+e^{\frac{1^0}{4}})$	0.99987	$2r(\pi-\frac{\pi}{r^{6^{\varphi}}})$	0.99999
5	$r(\pi+\pi+r^{-6^{r^r}})$	1.00000	$r(5+e^{\frac{1^{-1}}{4}})$	0.99987	$2r(\pi-\frac{1}{(6\times 6)^r})$	0.99999

Table 2 - Equations and associated fitnesses for the circumference of a circle

It is worth noting that the minimum radius in this data set was 3.2 – as such, $r^{-6/7}$ is negligible under these conditions. For radii less than one, this term would become increasingly large, making this a poor choice, but this can only be determined through comparison to the actual formula.

Volume of a Cuboid

Convergence Times

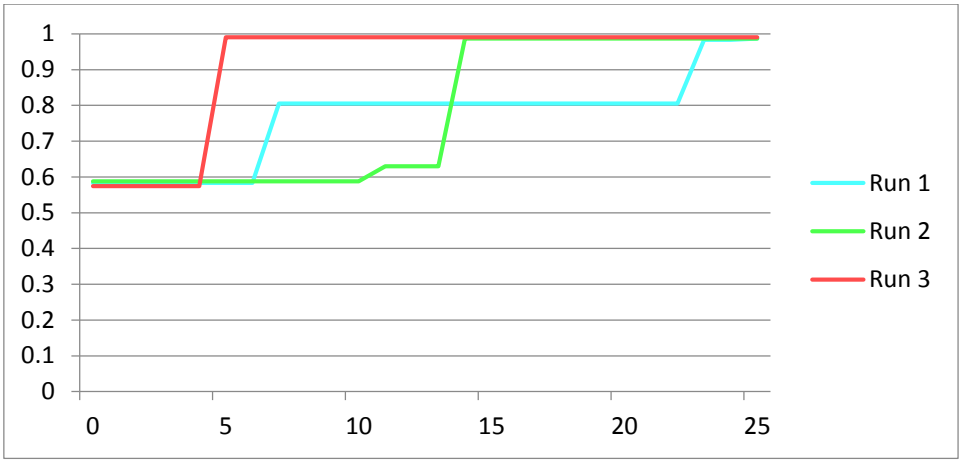

Figure 13 - Graph showing the convergence times for the volume of a cuboid

All three runs stabilised in 25 generations or fewer – it is just visible that the third run is marginally higher than the other two, due to the reduced number of elements in the final equation.

Final Output

Position	1st Run		2nd Run		3rd Run	
	Equation	Fitness	Equation	Fitness	Equation	Fitness
1	$\frac{d}{1 \div (w \times h)}$	1.00000	$\frac{w \times h}{1 \div d}$	1.00000	$h \times w \times d$	1.00000
2	$(w \times d)^{[1]} \times h$	1.00000	$\frac{w \times h}{[1] \div d}$	1.00000	$h \times d^{[1]} \times w$	1.00000
3	$(w \times d)^{[1]} \times h$	1.00000	$\frac{w \times h}{[1] \div d}$	1.00000	$h \times d^{[1]} \times w$	1.00000
4	$(w \times d)^{[1]} \times h$	1.00000	$\frac{w \times h}{[1] \div d}$	1.00000	$h \times d^{[1]} \times w$	1.00000
5	$(w \times d)^{[1]} \times h$	1.00000	$\frac{w \times h}{[1] \div d}$	1.00000	$h \times d^{[1]} \times w$	1.00000

Table 3 - Equations and associated fitnesses for the volume of a cuboid

A very good set of results here – a lack of constants makes the program much more effective, because there is nothing for it to approximate. This is the first instance of [1] indicating a

complex expression that is always 1. In every instance here, the expression is X^0, as this is the most likely to still be 1 after mutation. Any instance of $\frac{X}{X}$, for instance, can be upset by mutating either top or bottom.

Volume of a Square-Based Pyramid

Convergence Times

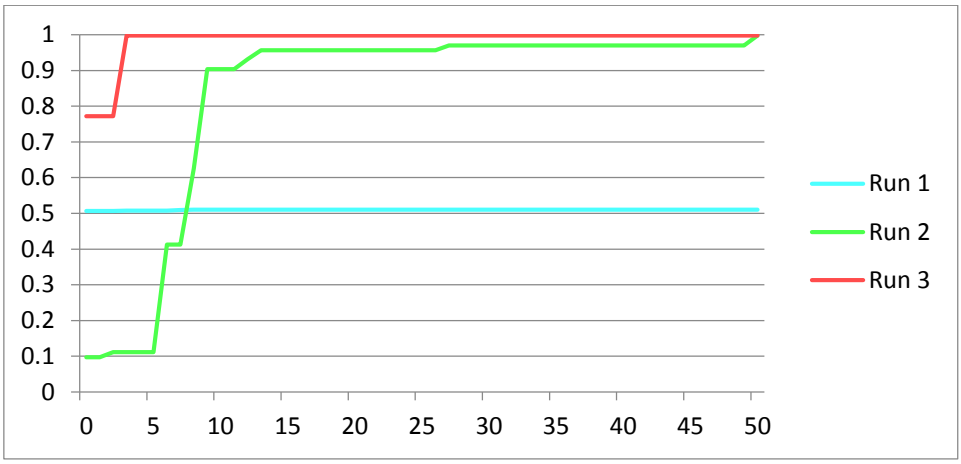

Figure 14 - Graph showing the convergence times for the volume of a pyramid

This graph shows that the maximum time taken to settle at a result was 50 generations. We can also see that the first run was stable – it managed to find a local maximum from the criteria given, and was unable to escape.

Final Output

Position	1st Run		2nd Run		3rd Run	
	Equation	Fitness	Equation	Fitness	Equation	Fitness
1	$\frac{b}{12 \times 11^{11}}$	0.00000	$\frac{b \times b}{3 \div h}$	1.00000	$\frac{b}{(3 \div b) \div h}$	1.00000
2	$\frac{h}{9 \times 11^{11}}$	0.00000	$\frac{b \times b}{[1] \times \pi \div h}$	0.95280	$\frac{b}{\left(3 \div ([0] + b)\right) \div h}$	1.00000
3	$\frac{h}{8 \times 11^{11}}$	0.00000	$\frac{b \times b}{[1] \times \pi \div h}$	0.95280	$\frac{b}{\left(3 \div ([0] + b)\right) \div h}$	1.00000
4	$\frac{b}{7 \times 11^{11}}$	0.00000	$\frac{b \times b}{[1] \times \pi \div h}$	0.95280	$\frac{b}{\left(3 \div ([0] + b)\right) \div h}$	1.00000
5	$\frac{h}{7 \times 11^{11}}$	0.00000	$\frac{b \times b}{[1] \times \pi \div h}$	0.95280	$\frac{b}{\left(3 \div ([0] + b)\right) \div h}$	1.00000

Table 4 - Equations and associated fitnesses for the volume of a pyramid

The first run has failed, by approximating all of its results to (almost) 0. This satisfies one of the fitness metrics – that the equation should be as good for all points in the data-set. This indicates two things: the cut-off value (how many orders of magnitude away the result can be) should be set to less than 1; and the weighting of the spread metric should be revised downwards. However, all three were running under the same parameters, so this is not guaranteed to fail as it is. In the second run, [1] indicates that there was a complicated section that always evaluated to 1, eg. $\left(7^{\frac{e}{3\varphi - b}}\right)^{0}$. As such, it was able to produce the same result an arbitrary number of times by altering the terms within the 0 power. In the third run, [0] indicates a complicated section that evaluated to (almost) 0, eg. $5^{\frac{9-3^{8}}{b}}$. Again, it was able to generate multiple results by changing these terms.

Volume of a Torus

Convergence Times

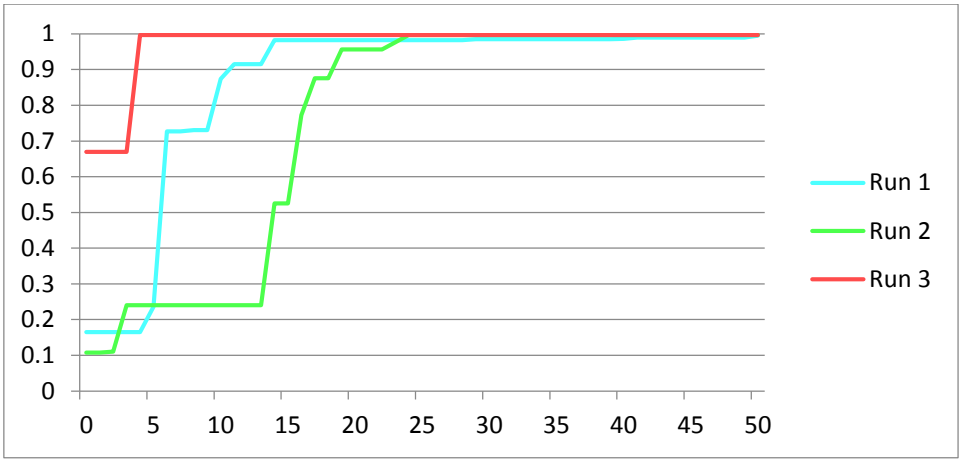

Figure 15 - Graph showing the convergence times for the volume of a torus

This run achieved stability after 50 generations, as with the square-based pyramid, but with all three finding a suitable answer. The first run can be seen slightly below the other two, due to being slightly more complex.

Final Output

The data set as originally computed was in fact for $\frac{\text{volume}}{\pi}$ due to an oversight, so all equations are evaluated on the assumption that they are missing a factor of π.

Position	1st Run		2nd Run		3rd Run	
	Equation	Fitness	Equation	Fitness	Equation	Fitness
1	$Rrr3\left(\pi-\frac{\pi}{3}\right)$	1.00000	$rr\pi 2R$	1.00000	$r2rR\pi$	1.00000
2	$Rrr3\left(e-\frac{1}{e-9^{\varphi\div 12e}}\right)$	0.99995	$rrR\left(\frac{\pi}{\varphi([1]+e+\pi)}+6\right)$	0.99998	$rrR(\frac{\pi}{e(1+\pi)}+6)$	0.99934
3	$Rrr3\left(e-\frac{1}{e-6^{e\div(42+e)}}\right)$	0.99993	$rrR\left(\frac{\pi}{\varphi([1]+e+\pi)}+6\right)$	0.99998	$rrR(\frac{e}{2(2+e)}+6)$	0.99922
4	$Rrr3\left(e-\frac{1}{e-6^{e\div 45}}\right)$	0.99979	$rrR\left(\frac{\pi}{\varphi([1]+e+\pi)}+6\right)$	0.99998	$rrR(\frac{5}{18}+6)$	0.99914
5	$Rrr3\left(e-\frac{1}{e-7^{\pi\div(54+e)}}\right)$	0.99795	$\left(\frac{3}{\pi(\varphi\div(4+\pi)+\pi)}+6\right)rrR$	0.99995	$rrR(\frac{10}{11\pi}+6)$	0.99902

Table 5 - Equations and associated fitnesses for the volume of a torus

As with the surface area of the torus, we have some interesting approximations for 2π being created. It is interesting that the second and third runs both approximate 2π as 'six-and-a-bit'. The second run utilises [1] as a placeholder for a more complex expression that is always equivalent to 1.

Results Analysis

From the results, we can determine firstly that the program is functional – in almost every case it found the correct solution, along with variations and approximations. In the one instance that the program failed, it was allowed to by a possibly poor choice of parameters. We can also see how complexity affects the run-time of the program – the fairly simple $2\pi r$ was either generated in the initial step, or arrived at within the first few generations, whereas the volume of a cuboid, involving three independent variables, required 25 generations and the volume of a torus, involving two independent variables, a power and a constant, took 50. Without doing further simulations with more complex data and more generations, it is difficult to estimate the level of scaling we can expect.

We can also see that for these relatively simple problems, a lot of the available parameters are not necessarily relevant. Many of these are likely to come into play if further work is done, as recommended, on more complex areas. Tuning the program will then be of more relevance, as the computational power required approaches the limits, and optimisation becomes necessary.

The tendency of the program to use approximations rather than the unity factor is dependent on the length boundary and the weighting given – the expressions which evaluate to 1 tend to balloon, which reduces the fitness of the equation – and whether or not the formula has a constant to be approximated. For the purpose of this report, all simulations had the same (low) weighting given to length as a factor.

Conclusions

Language Decision

The decision was made to use Erlang, despite unfamiliarity with it, and, having worked with it for over six months, I have had no problems with it. The documentation made available by Eriksson is comprehensive, and the language itself is consistent. No bugs have been found other than those introduced by myself. I would be surprised if MatLab or Python would have been so smooth, using external libraries that may or may not have been debugged properly.

Understanding of GP

I now have a fairly full understanding of Genetic Programming. I would by no means say that I have perfect knowledge on the subject, but I certainly understand the basics and some of the more complicated methodologies of the topic.

Code Writing

The code was finished near the end of Term 2, only slightly behind schedule. A large part of the code itself is visible in this report, and it can be seen that the function names are representative of their operation. The comments have been stripped out to save space in this report, given that each block of code has been explained beneath, but the full source code has comprehensive comments. At this point in time, the program is able to successfully solve data sets of at least medium complexity very regularly.

Presentation of Results

All output has been formatted to be slightly more readable, but retains the same basic structure as output by the program. The results are in two sections: an in-depth view of how the parameters affect the operation of the program, covering 96 different combinations, across six different parameters; and an overview of the output of five other data sets, showing the fitnesses and outputs, and the number of generations required to converge in each case.

Costing

Supervisor Time

14 × 30 minute meetings = 7 hours

7 hours × £50 / hour = £350

Technician Time

n/a

Student Time

300 hours × £15 / hour = £4500

Printing Costs

2 × 72 pages = 144 pages

144 pages × 20p / page = £28.80

Total Project Cost

£4878.80

Recommendation for Further Work

The project as documented here is entirely functional, but there are multiple ways in which it could be improved. Currently, the lack of Graphical User Interface combined with strict requirements on the format of the parameters file make it a very fragile system to those who are not versed in its eccentricities. Relaxing the formatting of the parameters file would produce a lot of code that compensates for possible user error, so a GUI would seem a more prudent solution, but there are two serious issues: A GUI leads to the possibility of a mismatch whereby an option removed from the program is still included in the GUI, leading to confusion for the user; and it becomes possible for a newly included parameter to be omitted from the GUI, and thus not be defined for the program running, leading to possible errors, crashes, and undefined output.

With regards to display, the current progress report is a countdown until the generational limit is reached. Showing the user the current best solution(s) and associated fitness(es) would allow the user to make an informed decision as to whether the simulation can be terminated early or should be allowed to run its course.

There are three improvements in functionality that I would recommend initially. Firstly, the single best result is carried from generation to generation, in tournament style. However, being able to define the depth of this carry and implementing a fuller tournament system would be desirable. Secondly, the language in which the program is written is designed for parallel processing across a cluster, and, at the time of writing, the program is restricted to the local machine. Extending the spawning process to utilise a wider environment would require a stable network (any lost processes currently stall the program completely) or compensation for that possibility, but could significantly reduce run-times by spreading the

load. Finally, the current system has no way of simplifying a final result. Unity gain, adding and subtracting the same variable, and spurious elements (such as adding insignificantly small values) all reduce the probability of identifying the best solution, as they are a numerical match for the dataset. By adding in a post-processing step to eliminate these possibilities the output becomes cleaner and more precise.

In terms of the analysis performed in this report, the program has been put through its paces to determine that it is functional, but has not been stretched. More complex data sets are beyond the scope of the current project, but should be considered for the future. Another useful feature to aid in analysis would be the logging of (or ability to log) all fitnesses, or an average fitness among the top N results.

Code of Ethics and Professional Conduct

All work not performed by myself has been referenced and attributed to the original authors. All data sets used were created personally. All code for this project has been written from scratch, or is included in the Erlang environment, and as such is covered by the Erlang Public License, which grants permission for usage, reproduction, modification, display, performance, sublicensing and distribution[3].

[3] http://www.erlang.org/EPLICENSE (accessed 20/04/14, 17:32)

Bibliography

Bäck, T., Hammel, U., & Schwefel, H.-P. (1997). Evolutionary Computation: Comments on the History and Current State. *IEEE Transactions on Evolutionary Computation*, 3-17.

Barricelli, N. A. (1957). *Symbiogenetic evolution processes realized by artificial methods.* Methodos.

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. *Proceedings of the First International Conference on Genetic Algorithms*, 183-187.

Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial intelligence through simulated evolution.* John Wiley and Sons.

Forrest, S., Nguyen, T., Weimer, W., & Le Goues, C. (2009). A genetic programming approach to automated software repair. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 947-954.

Jordaan, E., Kordon, A., Chiang, L., & Smits, G. (2004). Robust inferential sensors based on ensemble of predictors generated by genetic programming. *Parallel Problem Solving from Nature - PPSN VIII*, 522-531.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection.* MIT Press.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

Koza, J. R., Andre, D., Bennet, I. F., & Keane, M. (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufman.

Koza, J. R., Bennett, I. F., & Stiffelman, O. (1999). *Genetic Programming as a Darwinian Invention Machine.* Berlin Heidelberg: Springer.

Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Lanza, G., & Yu, J. (2006). *Genetic programming IV: Routine Human-Competitive Machine Intelligence.* Springer.

Langdon, W. B., & Nordin, P. (2001). Evolving hand-eye coordination for a humanoid robot with machine code genetic programming. *Genetic Programming, Proceedings of EuroGP'2001*, 313-324.

Lohn, J., Hornby, G., & Linden, D. (2004). Evolutionary antenna design for a NASA spacecraft. In e. a. U.-M. O'Reilly, *Genetic Programming Theory and Practice II* (pp. 301-315). Ann Arbor: Springer.

Nakano, T., Eckford, A. W., & Haraguchi, T. (2013). *Molecular Communication.* Cambridge University Press.

Negnevitsky, M. (2005). *Artificial intelligence: a guide to intelligent systems.* Pearson Education.

Poli, R., Langdon, W. B., & McPhee, N. F. (2008). *A Field Guide to Genetic Programming.*

Spector, L. (2004). *Automatic Quantum Computer Programming: A Genetic Programming Approach.* Kluwer Academic.

Spector, L., Clark, D. M., Lindsay, I., Barr, B., & Klein, J. (2008). Genetic Programming for Finite Algebras. *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, 1291-1298.

Trujillo, L., & Olague, G. (2006). Using evolution to learn how to perform interest point detection. *ICPR 2006 18th International Conference on Pattern Recognition*, 211-214.

Appendix 1: Full Table of Results

Problem	Crossover	Mutation	Multiple Mutations	Duplicate Number	Refill Pool	Pool Size	Generations		
							1	2	3
Torus Surface Area	Low	Low	No	1	No	100	2	3	2
						1000	3	X	2
					Yes	100	3	2	3
						1000	2	3	2
				3	No	100	2	X	2
						1000	1	1	3
					Yes	100	2	1	X
						1000	2	1	3
				No Limit	No	100	2	3	3
						1000	X	1	3
					Yes	100	X	2	1
						1000	X	1	1
			Yes	1	No	100	3	2	3
						1000	2	3	X
					Yes	100	X	2	2
						1000	X	X	1
				3	No	100	3	X	3
						1000	3	3	3
					Yes	100	3	X	3
						1000	1	1	1
				No Limit	No	100	3	3	X
						1000	1	1	1
					Yes	100	2	X	3
						1000	1	3	3

Problem	Crossover	Mutation	Multiple Mutations	Duplicate Number	Refill Pool	Pool Size	Generations		
							1	2	3
Torus Surface Area	Low	High	No	1	No	100	3	3	2
						1000	3	3	X
					Yes	100	2	3	1
						1000	2	1	3
				3	No	100	3	2	1
						1000	2	1	2
					Yes	100	3	3	3
						1000	3	X	3
				No Limit	No	100	X	X	3
						1000	X	1	X
					Yes	100	X	3	X
						1000	1	X	X
			Yes	1	No	100	X	2	2
						1000	1	3	2
					Yes	100	2	X	3
						1000	3	3	3
				3	No	100	X	3	X
						1000	1	2	3
					Yes	100	2	3	3
						1000	X	1	3
				No Limit	No	100	X	X	3
						1000	1	3	2
					Yes	100	X	X	1
						1000	3	1	1

Problem	Crossover	Mutation	Multiple Mutations	Duplicate Number	Refill Pool	Pool Size	Generations		
							1	2	3
Torus Surface Area	High	Low	No	1	No	100	2	X	1
						1000	X	X	2
					Yes	100	1	2	3
						1000	1	3	3
				3	No	100	1	X	1
						1000	3	X	1
					Yes	100	3	X	X
						1000	2	1	1
				No Limit	No	100	X	3	3
						1000	3	2	1
					Yes	100	X	X	X
						1000	3	2	3
			Yes	1	No	100	3	3	2
						1000	X	3	1
					Yes	100	3	3	3
						1000	X	2	2
				3	No	100	X	3	X
						1000	3	3	X
					Yes	100	2	2	3
						1000	3	X	3
				No Limit	No	100	3	X	3
						1000	X	3	3
					Yes	100	X	3	2
						1000	X	2	X

Problem	Crossover	Mutation	Multiple Mutations	Duplicate Number	Refill Pool	Pool Size	Generations		
							1	2	3
Torus Surface Area	High	High	No	1	No	100	2	3	1
						1000	X	2	2
					Yes	100	X	X	X
						1000	X	3	1
				3	No	100	3	1	2
						1000	3	3	2
					Yes	100	3	3	X
						1000	3	3	3
				No Limit	No	100	X	X	3
						1000	2	3	3
					Yes	100	3	2	X
						1000	X	3	3
			Yes	1	No	100	3	2	3
						1000	2	3	3
					Yes	100	2	2	2
						1000	3	3	3
				3	No	100	3	3	X
						1000	2	3	2
					Yes	100	2	3	2
						1000	3	X	1
				No Limit	No	100	3	3	3
						1000	2	1	3
					Yes	100	3	3	1
						1000	3	2	3

Appendix 2: Commit Log and Statements

Thu Mar 20 17:19:03 2014 +0000
Ability to Change Directory
We now change directory so that the data etc. can be stored more logically, and access by relative paths.

Thu Mar 20 17:17:25 2014 +0000
Removing Chaff
Data etc. moved to Dropbox, removed from repo.

Thu Mar 6 11:59:28 2014 +0000
Tab Alignment Fix
Some functions were misaligned - no more!

Tue Mar 4 16:56:53 2014 +0000
Equation Assemble Error
Now handles an incorrect equation in the pool file, rather than error-ing.

Thu Feb 20 00:05:11 2014 +0000
Best Result Hangover
Best Result from each generation now definitely carried over, so that the best result can't be lost.

Wed Feb 19 23:56:24 2014 +0000
String to Float Abstracted
Abstracted String to Float to my own function to include integers.

Tue Feb 18 02:32:58 2014 +0000
Collapsed Parameters, Cancellation
Collapsed down parameters for easier passing.
Spawned a cancel monitor which leaves a floating prompt offering the opportunity to cancel out at any time, and see results as at that point.

Tue Feb 18 02:31:29 2014 +0000
Added New Parameters, Cleared Pool
Parameters are now up to date with code.
Pool is clear for reasons.

Tue Feb 18 02:30:55 2014 +0000
Parameter Collapse, Output Trim
Parameters have been collapsed slightly.
Equations with fitness 0 are no longer written to the output, due to being uninteresting.

Tue Feb 18 02:30:02 2014 +0000
Collapsed Parameters, Abstracted Receiver, Added Cancel, Remove Duplicates, Refill Pool, Multiple Mutation
Parameters have been collapsed into tuples for passing.
Receiver has been abstracted to allow for possible recursive calls.
Cancellation is now possible - current generation will complete then file will be written.
Duplicates beyond a certain number will now be removed to maintain the integrity of the pool.
Pool now has the option of being refilled with randomly generated equations to stay 'fresh'.
Mutations now have the chance to 're-mutate' one or more times.

Tue Feb 18 02:25:55 2014 +0000
Collapsed Parameters for Easier Modification
Parameters are now collapsed into tuples so that they can be passed through the system to where they're needed as a group, rather than having to be modified in every intermediate location.

Mon Feb 17 00:59:06 2014 +0000
Display, OutMode, MaxDepth
Now displays when the setup is done, sets up the generation counter and announces when finished.
Reads in and passes through the overwrite/append state for the output files.
Initial pool generation now has set maximum depth to prevent out-of-control equation generation killing the memory.

Mon Feb 17 00:56:17 2014 +0000
Corrections
Changed data file to be floats which was breaking all the things.
Changed pool file to have the answer as a test.
Changed parameters to group output files at the end and add the outmode / maxDepth.

Mon Feb 17 00:55:03 2014 +0000
Mode, Floats and Display
Now have the option to overwrite or append the output files.
Outputs are now multiplied by 1.0 so that float_to_list doesn't get an integer and flip out.

Mon Feb 17 00:52:53 2014 +0000
Consolidation and Rework + Display
Now rejects any records that miss any criteria rather than just weighted criteria, and consolidated to keep logic in one place to avoid half-changes.
Added a display for generations left to stdout.

Mon Feb 17 00:50:46 2014 +0000
Evaluator Non-Zero
The evaluator now rejects answers of 0. They skew the spread analysis, have a very fixed miss magnitude and will screw up elsewhere. Marked as invalid.

Thu Feb 13 18:31:27 2014 +0000
Pool Generation Adapted
Pool generation adapted to be marginally less memory intensive.

Thu Feb 13 18:28:44 2014 +0000
Add Output Files to .gitignore
Changed output files to .out.csv for easier recognition, and added to ignore.

Thu Feb 13 18:27:33 2014 +0000
Clear Pool
Just because...

Tue Feb 11 21:25:24 2014 +0000
Updated Parameters + Change of Data
Parameters now include tracking file and weightings.
Data now split into volume and circumference, for variety.

Tue Feb 11 21:22:52 2014 +0000
Output Split
Output is now split into two files - Tracking and Output.
Tracking has the best fitness from each generation for graphing progress.
Output has the final pool and respective fitnesses in order from best to worst.

Tue Feb 11 21:21:56 2014 +0000
Fitness Weighting + List Reversal
Fitnesses are now calculated using weightings and cutoffs from the parameters file.
Lists were interpreted the wrong way round - best result is last, not first as was used beforehand.

Tue Feb 11 21:20:37 2014 +0000
New Output File + New Settings
Fitness weighting and cutoffs are now read from the parameters file.
Generation fitnesses now stored in tracking file, with pool dumped to output on finish.

Tue Feb 11 21:19:19 2014 +0000
Upgrade to Evaluation
Evaluator now returns the fitness value, the spread of values and the length of the equation for processing.

Thu Jan 30 16:36:53 2014 +0000
Output
Now outputs final best result and list of highest fitness values to file specified in testparameters.csv. File must be sans path for now.

Thu Jan 30 16:35:41 2014 +0000
Record of progress
Best result in each generation now recorded for output at end, for tracking.

Tue Jan 21 17:54:50 2014 +0000
Stochastic Implemented
Stochastic implemented - point choosing abstracted further up to allow for stochastic to utilise previous code.

Tue Jan 21 17:28:10 2014 +0000
Introduce Stochastic vs Roulette, Even Crossover Rate
Two methods of running - stochastic and roulette. Only change here is calling different functions in operation.
If crossover is odd, steal one from clone to avoid throwing stochastic off.

Tue Jan 21 17:26:08 2014 +0000
Output Creation
Procedure to organise equations into printable form.

Fri Jan 10 23:43:34 2014 +0000
Messaging, Standardising and Integers
Messaging and new generations now in place.
Standardising now tesseracts the value, to better differentiate the top end of equations.
Now using integers in equations, rather then arbitrary floats. Mutation does +-1 now.

Fri Jan 10 23:30:56 2014 +0000
New Procedure 'Run'
Gets the parameters and launches into the progamme.

Fri Jan 10 23:25:42 2014 +0000
Change run conditions
Less clone, more crossover and mutate, longer running and larger pool, no longer giving it the answer at the start.

Fri Jan 10 23:21:09 2014 +0000
Remove -compile(debug_info)
It does nothing!

Fri Jan 10 23:18:24 2014 +0000
Evaluator Refactor
Remove if and replace with guards.

Sun Dec 29 00:46:20 2013 +0000
Adding Catch Functionality
Value is now 'caught', so in the event of a rogue function (/0, complex outcome, etc.) the equation is marked as invalid rather than giving an error.
Filereader re-edited to include straight power and divide, to avoid watering down.
Operation now filters out all invalid functions before creating the new generation.

Sun Dec 1 21:14:51 2013 +0000
Test Changes
Changed testfunctions to reflect removal of ^ and addition of ^2 and ^3.
Added Generations to testparameters.

Sun Dec 1 21:13:58 2013 +0000
Standard Functions
Made Standard Float to List conversion to centrally control limit on decimal places in variables (currently 10).

Sun Dec 1 21:13:13 2013 +0000
Sorting and Standardisation
Code to sort pool and standardise between 1 and 0 now in place. New generation code altered slightly to heed a compiler warning, and to leave a better place for messaging.

Sun Dec 1 21:11:56 2013 +0000
Generated Pool Sorted and Standardised
The pool that is read is now vetted for validity.
The entire pool is now evaluated as it is generated.
The pool is then sorted to have best results at the top, standardised such that a good result is close to 1, and a bad result is close to 0.

Sun Dec 1 21:09:52 2013 +0000
/ and ^ Fix
/ had no guard against /0.
^ had no guard against complex numbers.
Added guard for /, implemented ^2 and ^3 as fixes to prevent complexity.
Sun Dec 1 21:08:20 2013 +0000
Syntax and Wording
Variable left behind from previous revision removed, + copy-paste errors fixed.

Sun Dec 1 21:07:12 2013 +0000
Delete Sticky Files
Random and Lists are in the main Erlang bin, so do not need local copies.

Sat Nov 30 01:38:43 2013 +0000
Fitness Evaluation Fix
Invalid equations and perfect equations were both returning 0. Now invalid returns the atom 'invalid'.

Thu Nov 28 15:55:49 2013 +0000
Pool Generation
Pool is now read from file, then additional equations generated until the pool size is reached. Alternately, if more equations than pool size are created, pool will be reduced to given size after first generation.

Thu Nov 28 15:54:09 2013 +0000
Merge branch 'master' of https://github.com/JamesAshworth/Erlang-GP

Thu Nov 28 15:53:52 2013 +0000
Filereader
Changed format of function file - now multiple rows of comma-separated.
Changed format of data file - now standard csv, w/ headings across top.
Changed format of pool file - now space separated polish notation, for readability.

Mon Nov 25 23:50:28 2013 +0000
Fitness + Variable Mutation
Fitness now goes for maximum deviation over average deviation.
Variable mutation now uses phi for symmetry, otherwise variables will tend to 0.

Sat Nov 23 23:53:27 2013 +0000
Evaluation, Generations
Evaluation rewritten and abstracted slightly
Operations abstracted to be behind generations

Sat Nov 23 17:58:54 2013 +0000
Mutation and Cloning
Mutating now works, and settings can be retrieved from the master settings function return.
Clone is also implemented.

Tue Nov 19 23:56:46 2013 +0000
Create README.md

Tue Nov 19 23:53:57 2013 +0000
File Reader
Pool file can now be read
Integer constants do not need to be defined as constants

Tue Nov 12 17:58:51 2013 +0000
File Reader, Settings Initialisation and Shit
Dealing with an initialised pool and reading parameters in.
POOL READING NOT DONE YET - TBD.

Sat Nov 9 19:17:25 2013 +0000
File Reader + Commenting
File reader now reads in constant and function files.
All code now commented in one way or another.
Demo file removed as superfluous.

Sat Nov 9 01:30:38 2013 +0000
File Reader
Added a read_file function to read in a csv file in a specified format and break it down into data tuples as required elsewhere – returns {[list of data tuples], number of data points}. Also added testdata illustrating circumference of a circle for 5 points for test purposes.

Wed Nov 6 00:42:16 2013 +0000
Comments
Comments file for making notes on chosen syntax etc.

Wed Nov 6 00:41:49 2013 +0000
Evaluator
Evaluator now deals with data and single argument functions

Tue Nov 5 18:05:34 2013 +0000
Evaluator
Initial attempt at an evaluator for binary tree functions.

Fri Nov 1 01:44:24 2013 +0000
Remove Ignored files from Repo

Fri Nov 1 01:43:26 2013 +0000
Ignore File

Fri Nov 1 01:39:20 2013 +0000
Initial Commit (Crossover)
Initial Commit to Git
Crossover already implemented in operation.erl