Genetic Programming

Iris Soute

Report number: 2000.25
September 2000

Table of Contents

List of Symbols

α	slope of sigmoid function	$[-]$
θ	position	$[\mathrm{rad}]$
$\dot{\theta}$	velocity	$\left[\frac{\mathrm{rad}}{\mathrm{s}}\right]$
$\ddot{\theta}$	acceleration	$\left[\frac{\mathrm{rad}}{\mathrm{s}^2}\right]$
$\sigma(x)$	sigmoid function	
$a(i)$	adjusted fitness	$[-]$
B_c	coulomb frictional coefficient	$\left[\frac{\mathrm{kg\,m}^2}{\mathrm{s}^2}\right]$
B_v	viscous frictional coefficient	$\left[\frac{\mathrm{kg\,m}^2}{\mathrm{s}}\right]$
c_m	motor constant	$\left[\frac{\mathrm{N\,m}}{\mathrm{V}}\right]$
F	function set	
$f(i)$	assigned fitness value	$[-]$
J	mass moment of inertia	$[\mathrm{kg\,m}^2]$
$n(i)$	normalized fitness	$[-]$
$p(i)$	probability	$[-]$
$r(i)$	raw fitness	$[-]$
$s(i)$	standard fitness	$[-]$
T	terminal set	
u	input signal	$[\mathrm{V}]$

Chapter 1
Introduction

Many potential inventions are never discovered because the thought processes of scientists and engineers are channeled along well-traveled paths. In contrast, the evolutionary process tends to opportunistically solve problems without considering whether the evolved solution comports with human preconceptions about whether the goal is impossible.

Most identification and control problems are complex and non-linear. Taking the evolutionary process as an example, solutions can be found the evolutionary way, by trial-and-error and the 'survival of the fittest'-principle, without the necessity of prior knowledge of a system or knowledge of (sometimes) complicated identification and control theories.

In 1948 Turing [1] saw the possibility of employing evolutionary and natural selection to create solutions. In the 1975 Holland [2] implemented Genetic Algorithms (GA), which can be seen as a predecessor of Genetic Programming. In 1992 Koza publicized his book "Genetic Programming: on the programming of computers by means of natural selection" [3] and thereby giving this new research field its name: Genetic Programming (or in short GP). Since then the interest in GP has grown rapidly.

The assignment was to explore the possibilities of GP, in particular for identification applications. In the first part of this report, the concept of GP will be further explained. Then, in the second part, the first results of GP runs for identification purposes will be evaluated.

Chapter 2
Genetic programming?

Genetic programming can be categorized as a form of Artificial Intelligence (AI), Machine Learning (ML), or Evolutionary Computing (EC). In all of these research field, scientists try to make computers 'intelligent' or 'self-learning'. Why? Because up until recently every computer-program written, was handmade. While hardware speed is exponentially getting faster, software development is not. Every line of code has to be written by a programmer which is very time-consuming. So, for several years, scientists have been trying to automate this process. How can computers learn to solve problems without being explicitly told how to? The existing methods of machine learning, AI, neural networks, etc., do find solutions in an 'intelligent' way, but the solutions are not represented in a convenient way. This is where GP differs from all methods named above: it finds a solution in the form of a computer program, which is executable. A GP algorithm works on a population of individuals, each of which represent a potential solution to the problem. GP uses the following steps:

1. Generate an initial population of random compositions of the *functions* and *terminals* of the problem (computer programs).
2. Execute each program in the population and assign it a *fitness* value according to how well it solves the problem.
3. Create new computer programs, also called offspring
   a. Copy the best existing programs (also called *reproduction*)
   b. Create new programs by *mutation*
   c. Create new programs by *crossover*
4. The best computer program that appeared in any generation, the best-so-far solution, is designated as the result of a GP-run.

A flowchart of the steps in GP as described above is shown in figure 2.1.

2.1 Representation

In most cases the individuals in a populations (the programs) are represented in a tree structure. For example the formula

$$y = \frac{a - b}{3} \tag{1}$$

can be presented as in figure 2.2.The smallest part of a tree is called a node, connected nodes are called a branch, see figure 2.2. The depth of a node is the minimal number of nodes that must be traversed to get from the root node of the tree to the selected node . The most left node in figure 2.2 has depth 3 [4]. The tree form is often used to display formulas in GP because it facilitates the use of genetic operations, as will become clearer later on.

2.2 Fitness & Selection

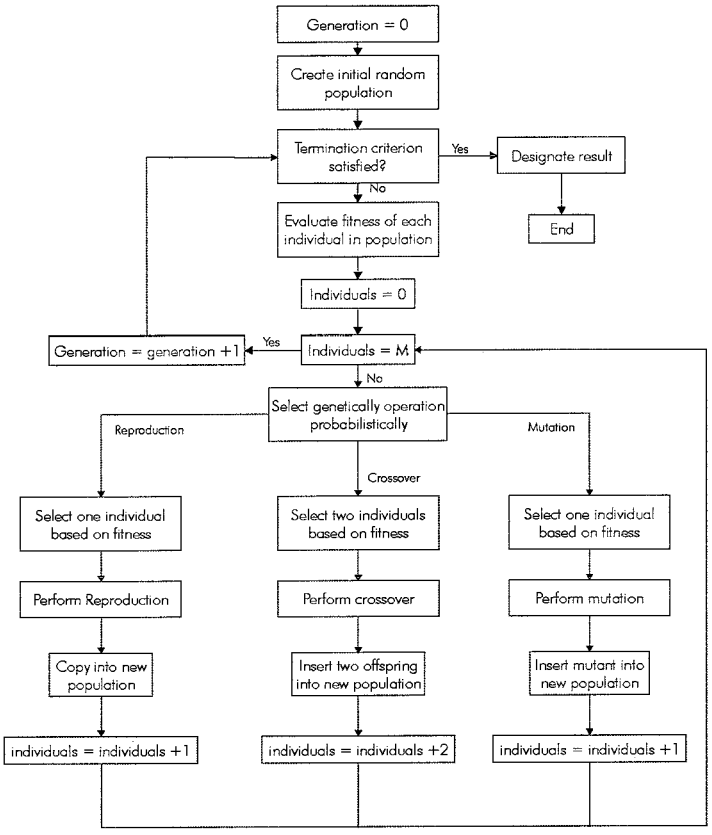

Figure 2.1: Flowchart of GP

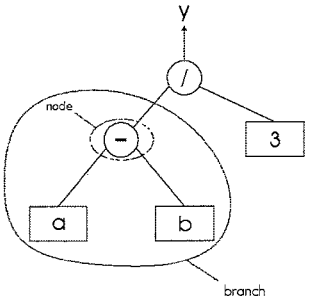

Figure 2.2: Tree-form representation of equation (1)

2.2.1 Fitness

The most difficult and most important concept of GP is the fitness function. Because this function determines how well an individual is able to solve the problem, fitness functions are very problem specific.

The fitness function gives feedback to the GP algorithm regarding which individuals of the population should have higher probability to be allowed to crossover, mutate or reproduce and which individuals should have higher probability to be removed from the population. The fitness functions assigns numeric

values to the individuals to provide a measure of the appropriateness of a solution. Also a fitness functions needs to be able to distinguish a more successful solution from a less successful solution.

Fitness is usually computed over a number of fitness cases. These fitness cases form the basis for evaluating the individuals in a population. The number of fitness cases should be sufficiently large as to produce a range of different numerical (raw) fitness values. The fitness cases are typically only a small finite sample of the entire domain space of interest, but should be representative of the domain space as a whole, because they form the basis for generalizing the results obtained to the entire domain space.

Different forms of fitness functions are:

- Raw fitness

  The raw fitness $r(i)$ is very problem specific. Therefor it is also problem specific what fitness value is a 'good' value. For one problem it could be that the larger $r(i)$ the better, for another problem it could be the other way around. In general, the raw fitness can be defined as the sum of errors between the solution of an individual and the solution given by the fitness cases. As the goal is to minimize the error, a small value (or even zero) is the desired value of fitness.

$$r(i) = \sum_{j=1}^{N_f} |S(i,j) - C(j)| \tag{2}$$

  where $S(i,j)$ is the value returned by the individual i for fitness case j (of N_f cases) and $C(j)$ is the desired value for the fitness case j. For symbolic regression problems, the raw fitness can also be the sum of the squares of the errors, so the influence (penalties) of distance points is larger.

- Standardized fitness

  The standardized fitness $s(i)$ recalculates the raw fitness so that the value assigned to the best individual is zero. If for a particular problem a lesser value of raw fitness is better, the standardized fitness equals the raw fitness.

- Adjusted fitness

  The adjusted fitness $a(i)$ can be calculated from the standardized fitness in the following way:

$$a(i) = \frac{1}{1 + s(i)} \tag{3}$$

  The value of the adjusted fitness always lies between 0 and 1, and the better the individual the bigger the value. The main benefit of the adjusted fitness is that it exaggerates the importance of small differences in the value of the standardized fitness as the standardized fitness approaches 0, which often occurs in later generations of a run.

- Normalized fitness

  The value of the normalized fitness $n(i)$, as the adjusted fitness, is a value between 0 and 1. The difference is that the sum of the normalized fitness values equals 1. Better individuals of the population have a larger normalized fitness value. It can be calculated from the adjusted fitness as follows:

$$n(i) = \frac{a(i)}{\sum_{k=1}^{M} a(k)} \tag{4}$$

2.2.2 Selection

After the fitness of the individuals in a population is assessed, it must be determined which individuals will selected to be subjected to the different GP operations to produce new individuals. Several selection methods are available:

- Fitness proportionate

  A individual is given a probability (to produce offspring) of

$$p_i = \frac{f_i}{\sum_i f_i} \tag{5}$$

  where f_i is the assigned fitness value.

- Rank selection

  Rank selection is based on the fitness order into which the individuals can be sorted. The selection probability is assigned to individuals as a function of their rank in the population.

- Tournament selection

  Tournament selection is not based on competition in the population as a whole. Instead, a small group of individuals is randomly chosen from the population. In this small group the tournament finds place: the better of the individuals are allowed to create offspring.

  Tournament selection has become one of the most popular selection methods, because it does not need a fitness evaluation of all the individuals, which reduces computing time considerably.

2.2.3 Generational versus steady state GP algorithm

There are two ways to execute a GP run. The first one is the generational GP run. During such a run, generation upon generation is created using for example the fitness proportionate selection method. From an old generation individuals are selected to create new individuals by crossover, mutation, or reproduction until a new generation is filled with these new individuals. The old generation is then discarded, and the process is started over again to produce a new generation, until some end criterion is met.

The other option is the steady state GP. No new generations are created now. The GP run starts to create a population with the specified number of individuals. Then, the tournament selection method is applied. Not all individuals are allowed to compete, just a small set of them, taken randomly from the population. Instead of putting the newly created individuals in a new generation, they are put back in the population, replacing the individuals that 'lost' during the tournament. Note that the 'winning' individuals are also returned to the population.

2.3 Functions and terminals

The terminal and function sets are the alphabet of the programs to be made. In fact, they represent the search space of a problem. The terminal set consists of variables (inputs) and constants of the programs. They are called terminals because they terminate or end a branch of a tree in a tree-based GP.

The function set is composed of the statements, operators, and functions available to the GP system. For example:

- Boolean functions
- Arithmetic functions
- Transcendental functions (trigonometric, logarithmic)
- Variable assignment functions
- Indexed memory functions
- Conditional statements

- Loop statements
- Control transfer statements

The functions and terminals chosen for a GP run, should be powerful enough to be able to solve the problem at hand. The smaller a function set is, the easier and faster it will be to find a solution, but only for simple problems. For more complex problems a larger function set is needed. One should be aware not to choose a set that is too large. A large function set enlarges the search space and will make the search for a solution harder.

Another important property of the function set is the closure property. Each function should be able to handle all values it might get as input. The most common example of a function that does not fulfill the closure property is the division operator. It cannot handle zero as an input. A solution is to define a new operator: the protected division. It acts like a normal division, except when it receives a zero as input. In that case it will return something else, a very large number or zero, for example.

2.4 Algorithm control parameters

The GP control parameters outline the way the GP run is executed. There are several parameters to be set before executing a GP run. A few examples:

- Termination criterion. This criterion prescribes when the run should stop. This is generally a pre-defined number of generations or an error tolerance on the fitness.
- Population size. This is the number of individuals in the population.
- Crossover-, mutation- and reproduction probabilities. These parameters control the degree of crossover, mutation and reproduction that will take place during the run. These parameters are often expressed in weighted values. A crossover probability of 0.7 and a mutation probability of 0.3 are often used values in GP-runs.
- Selection method. See section 2.2.
- Maximum individual size. This value refers to the maximum depth the individuals can obtain. When taking this parameter too large, the solutions will probably become too complicated, and computing time will go up. On the other hand, taking the parameter too small, can result in solutions that are too short to solve the whole problem.

2.5 Initialization

The first thing that is done when starting up a GP run is the initialization. During initialization the population is filled with individuals for later evolution. There are two ways to do this: *full* and *grow*. Grow produces trees of irregular shape because nodes are selected at random from the function and terminal set. Once a branch contains a terminal node, that branch is ended, even if the maximum depth has not been reached.

Instead of selecting nodes randomly, the full method selects only nodes from the function set until the maximum depth is reached. Then it selects only terminal nodes. The result is that all branches go till maximum depth [4].

2.6 Genetic operations

During a GP run, several operations are used on the individuals of the population to generate new generations. First an individual is selected from the population by means of the selection method described in paragraph 2.2. Then the operation that will be performed on the individual is chosen. All operations have an assigned probability value and this value corresponds with the probability that that operation will be chosen. More about these probabilities will be explained in paragraph 4.2. The most important operations will be explained here.

2.6.1 Crossover

The most important genetic operation in GP is the crossover operation. In the crossover operation, two solutions are combined to form two new solutions. The parents are chosen from the population by a function of the fitness as described in section 2.2. The crossover operation combines the properties of two parents by swapping a part of one parent with a part of the other, see figure 2.3.

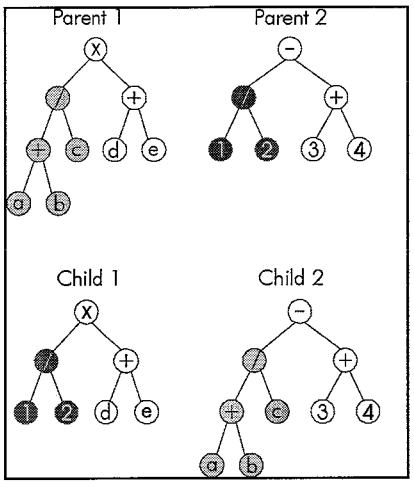

Figure 2.3: Example of crossover

2.6.2 Mutation

In the mutation operation, a single program is probabilistically selected from the population based on fitness.
Two types of mutation are possible:

1. a function can only replace a function or a terminal can only replace a terminal.
2. an entire subtree can replace another subtree.

A mutation point is chosen randomly, the function or subtree rooted at that point is deleted and a new function or subtree is grown there using the same random growth process that was used to generate the initial population, see figure 2.4.

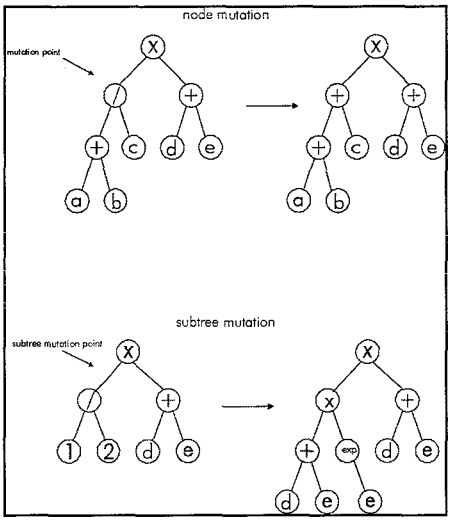

Figure 2.4: Examples of mutation

2.6.3 Reproduction or Copying

An individual is selected based on its fitness value and copied. The copy is placed into the population, which results in two versions of the same individual in the population

2.6.4 Automatically defined functions

An automatically defined function (ADF) is a function that is dynamically evolved during a run of genetic programming and that may be called by a calling program (or subprogram) that is concurrently being evolved. The program individual containing ADFs is a tree just like any program in regular tree-based GP. However, when using ADFs, a tree is divided into two parts or branches: one is the *result-producing branch,* which is evaluated during fitness calculation, and the other is the *function-defining branch,* which contains the definition of one or more ADFs [4] [5]. Because two different types of branches are used, it is not possible to use the simple crossover operation as described above. The crossover operation should be modified to make sure that only branches of the same type are used for crossover.

A weakness of the ADF approach is that the architecture (i.e. the two types of branches) of the overall program has to be defined beforehand. It would be much better if the complete structure of an individual could evolve including all ADF specifications. Koza [6] has proposed architecture altering operations as a method to achieve this goal. He posed six architecture altering genetic operations that can add initial ADF structures, clone them, and change the number of parameters.

Chapter 3
Applications & Software

3.1 Overview of applications of GP

Since the introduction of GP by Koza, a variety of applications have been researched. Here is an overview of the applications that have been published.

algorithms	control (robots and agents)	interactive evolution
art	control (spacecraft)	modeling
biotechnology	decision making	natural languages
computer graphics	electrical engineering	optimization
computing	financial prediction	pattern recognition
control (general)	hybrid systems	prediction
control (process)	image processing	signal processing

3.2 Software

The GP algorithm has been programmed using several different programming languages, including: C, C++, JAVA, LISP, and Prolog. Most programs are open source code and are available on the internet. Examples are:

- GPC++ by A. Fraser
- GP Quick (C++) by A. Singleton
- OMEGA, Predictive Modeling System by Cap Gemini
- Genetic programming studio (Lil-gp)
- Evolve (C++)

Commercial software is also available:

- Discipulus by Register Machine Learning Technologies Inc. Detailed information can be found at http://www.aimlearning.com.
- Genetic Search Toolbox for use with Matlab and Simulink by Optimal Synthesis Inc. Detailed information can be found at http://www.optisyn.com/gs/page1w.

Chapter 4
GP in practice

To test the possibilities of GP, I used the program GP Quick 2.1 written in C++ by Andy Singleton. Changes to the program were made, to accommodate the problems I wanted to solve, but the GP kernel itself stayed mainly as it was.

The fitness cases can be specified in a file. There is no limitation to how many fitness cases can be used. However obviously, the more points, the longer the calculation of fitness will take.

The results of the GP run are written to a file. At the start of a run, information about the run, like the function set used and other GP parameters, are written to this file. During the run, information about the 'best-so-far'-result is written to the file. The solutions and their fitness are also displayed on screen.

Solutions are displayed in prefix notation (see for explanation of prefix notation see appendix A) . A matlab program is used to convert the prefix solutions given by the GP program to infix notation.

The selection method used in all runs is the Tournament selection (thus steady state GP is applied), as this was the only form of selection provided by the program.

The program can handle arithmetic, exponential, goniometric and boolean functions but cannot generate ADFs.

All calculations were made using a computer with a pentium pro processor, 128 MB internal memory, running Microsoft Windows 95.

All control parameters were held at the same values during all the runs described below unless mentioned otherwise, with the exception of the population size and the number of fitness cases. Small problems don't need a large population size to converge fast to an accurate solution. Complex problems may need a larger population to obtain good results. Obviously there is a compromise between the speed and population size. A very large population size may be helpful to fully explore the search space, but needs much more time to compute. On the other hand: a very small population might not contain all elements for a perfect solution. The same goes for the number of fitness cases. As all the cases are evaluated to compute the fitness of an individual, it is easy to understand that there is a direct link between the number of fitness cases and the computing time needed.

4.1 Simple symbolic regression

Regression is often used to fit data to a curve. A function is pre-defined and with aid of the measured data, the unknown parameters are calculated. Using GP, it is possible to estimate not only the parameters, but also the symbolic form of the function. The only thing needed is data of input and output. GP will then map the input to the output.

Suppose we have obtained the following data:

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix}, y = \begin{bmatrix} 1 \\ 4 \\ 9 \\ 16 \\ 25 \\ 36 \\ 49 \\ 64 \\ 81 \\ 100 \end{bmatrix} \tag{6}$$

and we would like to fit a function $y = f(x)$. The steps that should be taken are as follows.

First we would define the terminal set. The output only depends on one input so the terminal set only contains x_1:

$$T = \{x_1\}$$

The terminal set should officially also contain a constant function. The constant function generates random (constant) values for use during the run. As the constant function is automatically inserted by GP Quick, we don't have to specify it explicitly in the terminal set.

Then the function set is to be defined. As the function to be fitted obviously is $y = x^2$, we would only need a multiplication function. Unfortunately, most problems are not as simple as this one, so a more general function set is chosen:

$$F = \{+, -, /, *\}$$

As problems get more complicated it is wise to add for example a sine and an exponential function to the set, to address a wider range of solutions.

The third step is to define a fitness function. For this problem a raw fitness is adequate:

$$f = \sum_{i=1}^{10} (y_i - GP(x_i))^2 \tag{7}$$

where $GP(x_i)$ is the result of the solution given by GP, evaluated at x_i.

The last preparatory step to be taken is to set an end criterion. In this case the problem is solved when the fitness has value 0. An alternative for an end criterion could be the number of programs created, or elapsed time.

After start-up of the run, the initialization takes place. The best-result obtained after initialization in this particular run is:

$$y = \frac{\frac{118}{x_1} + 81 - x_1}{197 - x_1} \tag{8}$$

In figure 4.1 the result is displayed.

After 50 individuals have been created, the best-so-far solution is:

$$y = \frac{\frac{-2975}{27} + 87x_1}{\frac{143}{31} + x_1 - \frac{1}{10x_1}} \tag{9}$$

which is plotted in figure 4.2.

After 1.26 [s] 70 individuals have been created and the exact result $y = x_1^2$ is found. In figure 4.3 the fitness of all the best individuals during the run is plotted.

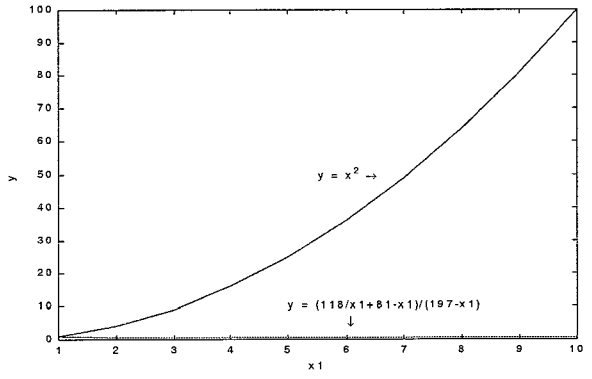

Figure 4.1: Best-of-run in initialization

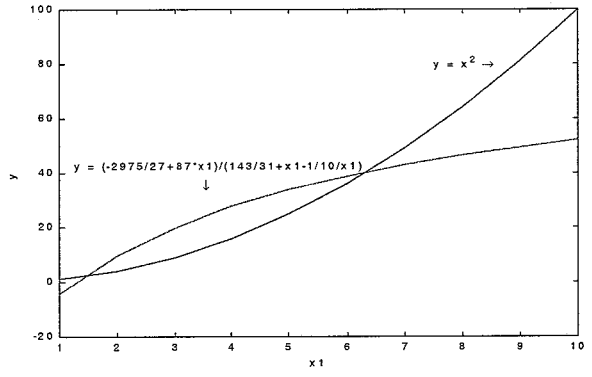

Figure 4.2: Best-so-far after 50 iterations

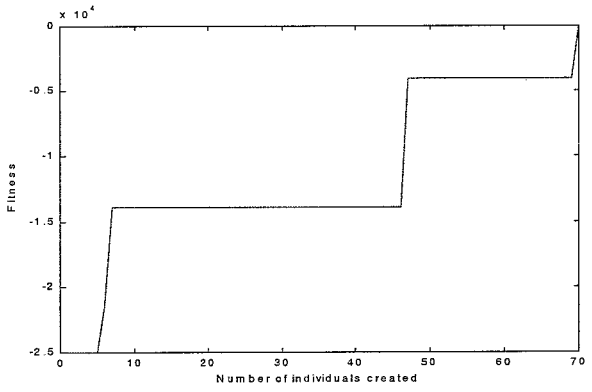

Figure 4.3: Best-of-run fitness values

When executing the same problem several times, the exact same solution is found everytime. The form of the (solution) tree may differ, but when simplified, the solution is the same. What really differs is

the computing time. To understand this take a look at the figure 4.4. All the trees represent the same formula, but contain a different number of nodes. The more nodes the GP algorithm has to traverse, the longer it takes. This is one reason why computing times can differ from run to run. Other reasons for differences in computing time will be discussed in the next paragraph.

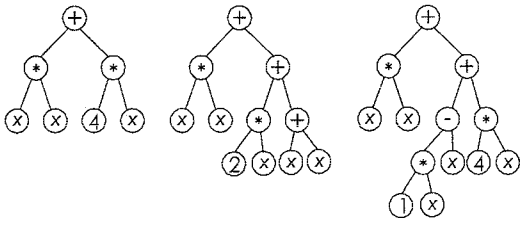

Figure 4.4: Same solutions, different tree forms

4.2 A simple model and the influence of control parameters

Research is being done on how to optimally select the control parameters for a run, but no real theoretical basis has been found yet. So, the user has to select the control parameters at random, or based on previous experience. To develop some sense in how the parameters influence a run a simple model is created to generate data. This data is then used to execute several runs, with different sets of parameters.

Using Simulink fitness cases are generated of the system:

$$u = Kx + B\dot{x} \tag{10}$$

taking $u = \sin(t)$, $B = 2$, and $K = 10$. The values of x, $\dot{x}$, and u are then given to the GP algorithm. This problem is in fact not much different from a symbolic regression problem with two variables. The only difference is that x and $\dot{x}$ are dependent variables. It is possible, in theory, to let GP estimate this dependency also, so that only x and u are needed for GP. To accomplish this, the GP program should have a differentiation function in the function set available. This is not (yet) possible in GP Quick and therefore the data is differentiated beforehand.

To test the influence of several control parameters this problem is evaluated one hundred times per value of a control parameter. The end criterion is set to 10.000 iterations (i.e. created individuals). By looking at the number of failed runs (i.e. the runs that did not come up with the perfect solution within 10.000 iterations), something can be said about the parameter. The mean time is the average time it took the good runs to convert to the perfect solution. The same goes for the mean number of iterations.

Number of fitness cases and population size

Population size	500	1000	3000	500	1000	3000	500	1000	3000
Number of fitness cases	50	50	50	200	200	200	1000	1000	1000
Number of failed runs	1	1	53	10	2	50	6	1	49
Mean time [s]	0.592	1.055	1.960	0.873	1.737	4.297	3.352	4.972	15.857
Mean number of iterations	2500	4102	8715	2947	4163	8378	2682	4218	8524

It can easily been seen and understood that the rise in time is proportional to the rise in population size and the rise in number of fitness cases. In both cases there are more individuals and/or fitness cases to be processed, which drives up the time. What strikes one most is the great number of failed runs at a population size of 3000. An explanation is that there are too much individuals to be processed for a

solution to emerge within the limit of 10.000 iterations. To test this, one more run was done but now the limit was set to 20.000 iterations, with the following results:

Population size	3000
Number of fitness cases	50
Number of failed runs	0
Mean time [s]	2.585
Mean number of iterations	9894

Now, all runs converged to a perfect solution. The average number of iterations needed to achieve lies very close to the first limit of 10.000 iterations. In figure 4.5 a histogram is drawn of the results of the hundred runs. Approximately half of the runs needed more than 10.000 iterations for a perfect result,

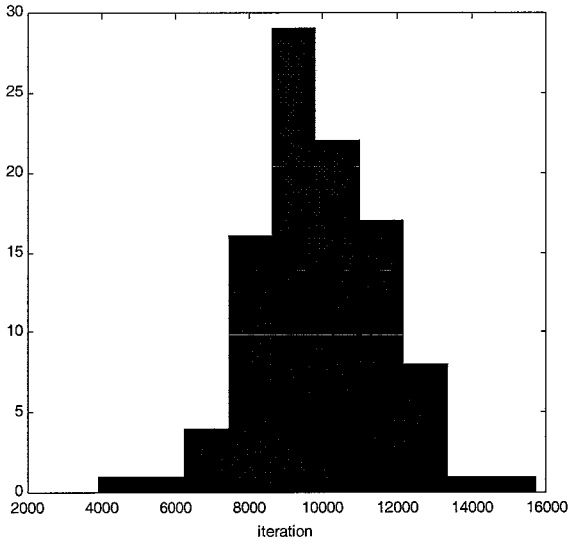

Figure 4.5: Iterations needed for perfect result (100 runs)

which is precisely why in the first test so many runs failed.

Some more tests pointed out that for this problem with as few as 15 fitness cases and a population size of 50 a perfect solution can be found, within a reasonable time. However, as can be expected, the number of failed runs also goes up, with decreasing values. So for the following tests the population size is set to 500 and the number of fitness cases to 50. As the average number of iterations needed for perfect result is 2500, the termination criterion is now set to 2500 iterations. This will speed up the calculations.

Note that these values are only valid for *this* problem. Every problem has its own optimal population size and optimal number of individuals needed to solve the problem. It is a process of trial and error to find these values. For some problems it might be worth the time to find these optimal values, as it will decrease computing time. On the other hand, as computer speed is still increasing, the solution will be found even though the parameters are chosen larger than optimal, without much influence on computing time.

Crossover probability

The crossover probability indicates the probability that the genetic operation being used is the crossover operation. Crossover is a heavily discussed operation, because it can greatly influence the speed and performance of a run. As one can imagine, individuals created by crossover tend to be far more different

from their parents then individuals created by mutation and reproduction. This characteristic feature is needed during a run to overcome a local suboptimal solution for example. On the other hand, research also pointed out that a large part of the offspring has a significantly lower fitness than their parents. A balance should be found between creating totally new individuals and destroying good traits in parents.

In fig. 4.6 the number of failed runs (of 100 runs) is displayed for different values of the crossover probability. As can be seen, a crossover probability of 0.7 is optimal for this problem.

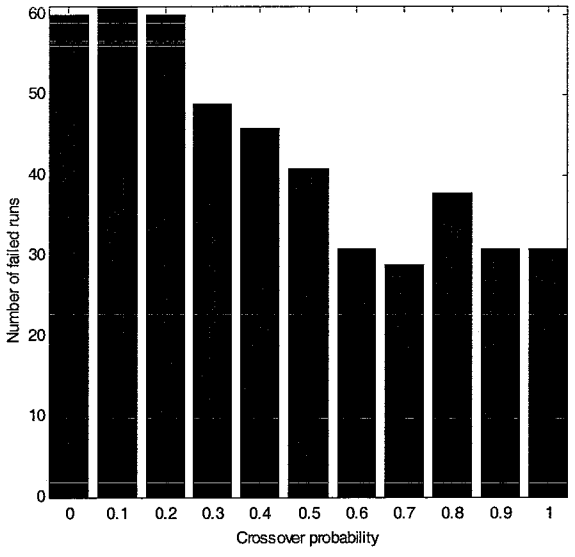

Figure 4.6: Influence of crossover probability

Mutation probability

The mutation operator has less influence than the crossover operation on the performance of a run. Fig. 4.7 indicates that a probability of 0.2 is optimal. In literature a combination of 0.7 crossover- and 0.3 mutation probability is often found.

Copy probability

The copy- or reproduction operation, as the mutation operation, has not much influence on the GP run. It doesn't create any new individuals, it only makes copies of good individuals. This is useful keeping in mind the destroying quality of the crossover operation. In figure 4.7 a value of 0.6 for the copy probability is the best value, although it doesn't differ much from the other values. When the copy probability is near to 1, the performance of the run becomes worse. This can easily be understood: as the copy operator becomes the main GP operation in a run, the chance of creating new individuals (by mutation or crossover) is very unlikely. Only runs that have (by accident) created the right solution during initialization succeed. Long runs will in the end also produce a good result, but in this case that is not likely to happen, because of the restraint of 2500 iterations.

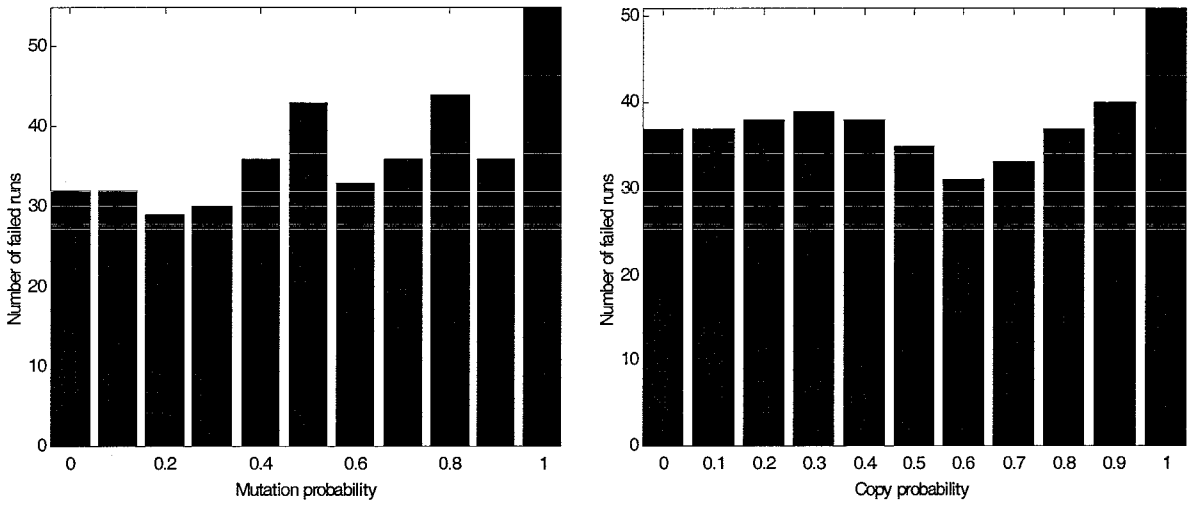

Figure 4.7: Influence of mutation and copy probability

4.3 Model of the pendulum

As the final goal of this project is to model the pendulum, with actual data from the pendulum, first a try-out is done with data generated by a model of the pendulum. The model used is:

$$\ddot{\theta} = \frac{c_m}{J}u - \frac{1}{J}\left(B_v\dot{\theta} + B_c\sigma\left(\dot{\theta}\right)\right) \tag{11}$$

with

$$\sigma\left(\dot{\theta}\right) = 1 - \frac{2}{e^{2\alpha\dot{\theta}} + 1} \tag{12}$$

with $J = 0.034$, $c_m = 16$, $B_v = 0.0554$, $B_c = 0.3164$, and $\alpha = 15$ equation (11) alters in:

$$\ddot{\theta} = 470.59u - 1.63\dot{\theta} - 9.31\sigma\left(\dot{\theta}\right) \tag{13}$$

As input u for the pendulum model a chirp signal was used. The corresponding value of θ was then calculated using Simulink. $\dot{\theta}$ and $\ddot{\theta}$ were zero-phase differentiated from θ. Then u, θ, $\dot{\theta}$, and $\ddot{\theta}$ were given to the GP algorithm.

Three runs were executed, each using a different function set.

First run

For the first run the following function set is used:

$$F = \{+, -, *, /\}$$

After 13680.9 [s] the best-so-far solution was:

$$\ddot{\theta} = \left(472 - \frac{u\dot{\theta}(8\dot{\theta} + 448 + 2\dot{\theta}^2)}{(u + \frac{1}{62}u^2(384 + 7\dot{\theta})\dot{\theta}^2 + 2)}\right)u \tag{14}$$

This model is simulated in Simulink. Comparing the results of the original model and the GP model gives the result plotted in figure 4.8. The results are fairly good, since not all functions that were used to create the data set were available during the GP run. The GP model was also tested with an input it was not trained for (step input), see figure 4.9.

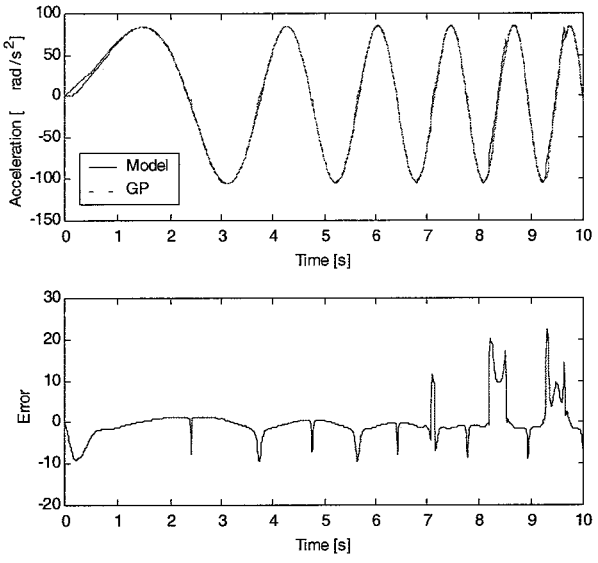

Figure 4.8: Response to chirp input (first run)

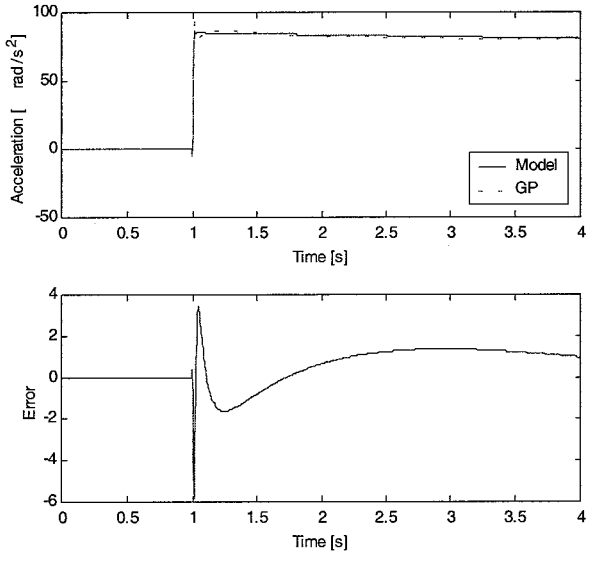

Figure 4.9: Response to step input (first run)

Second run

To extend the search space the exponential function is added to the function set for the second run. After 6426.13 [s] the result displayed in equation 15 and plotted in figure 4.10 is found.

$$\ddot{\theta} = \left(466 - \tfrac{1}{99}\theta(6u - 2u^2 - 3u^3\dot{\theta})\dot{\theta} - \tfrac{5}{33}\frac{\dot{\theta}}{u}\right) u \tag{15}$$

What is surprising is that the exponential function is *not* used in this result. In spite of this is the result somewhat better than in the previous run. During the run the exponential function has been used. It probably helped to overcome some local minimum in which the previous run got stuck. The results are shown in fig. 4.10. Due to numerical problems the step response of this model could not be calculated.

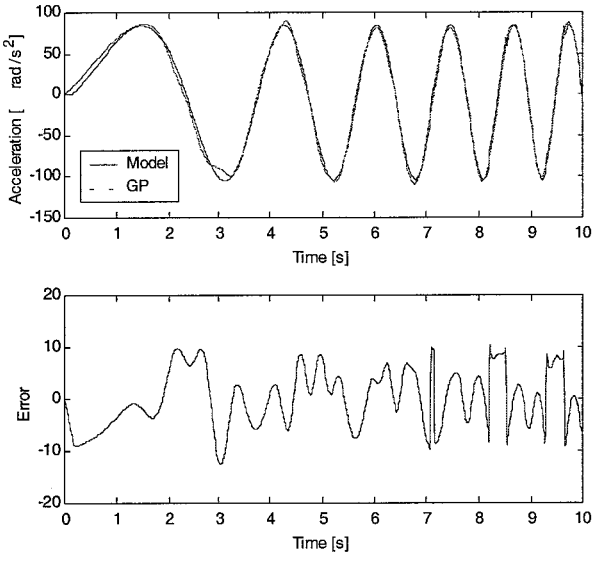

Figure 4.10: Response to chirp input (second run)

Third run

In the third run the exponential function is replaced by the sigmoid function as defined in equation (12). The result, equation (16), resembles the original model, equation (13) very much, two of the three terms have been estimated almost accurately.

$$\ddot{\theta} = 471.05u - 9\sigma\left(\dot{\theta}\right) - \sigma\left(\dot{\theta} - \sigma(\dot{\theta})\right) \tag{16}$$

The response to chirp- and step input of this model is shown in figure 4.11 and figure 4.12. The result is okay, except for the peak in the chirp response after about 7 seconds where a numerical problem manifests itself.

4.4 Data from the pendulum

Using a data set obtained from the pendulum, again three runs were executed to estimate a model. As input signal $u = 0.06\sin(t)$ was used. The position θ was measured and the velocity $\dot{\theta}$ and acceleration $\ddot{\theta}$ were calculated from θ using a zero-phase derivation function in Matlab. After that both signals

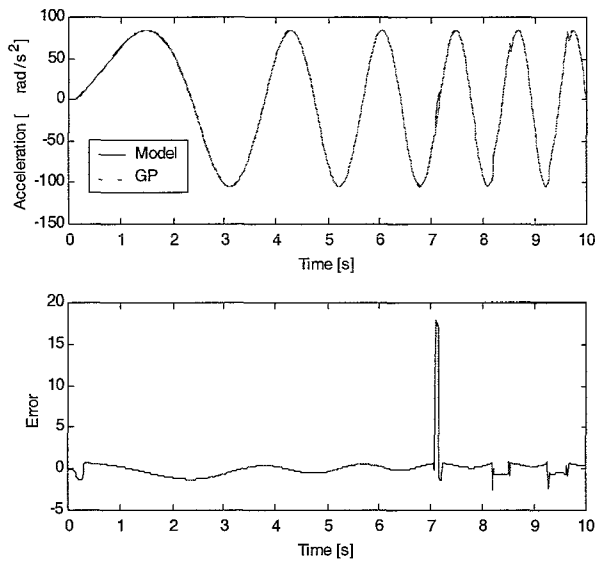

Figure 4.11: Response to chirp input (third run)

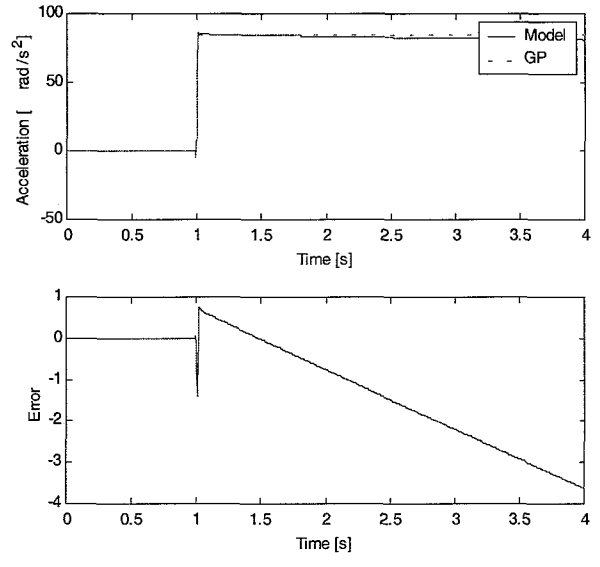

Figure 4.12: Response to step input (third run)

were filtered to remove noise with a zero-phase second-order lowpass Butterworth filter, with a cut-off frequency of $\omega_n = 20\,[\text{Hz}]$.

The population size was set to 3000 and 3000 fitness cases were used. The end criterium was zero error between the data and the GP model. Since this criterium was not met in any of the runs after (sometimes) several days of computing, the runs were stopped manually. Although the fitness of the best individual in the run was not significantly improving by then, it is not said that if the runs were continued for more

days that a better solution would not emerge. But as convergence rate to a better solution was very slow, the runs were stopped and the best models are presented here.

First run

The function set of the first run is made up as follows:

$$F = \{+, -, *, /\}$$

After 350105 [s] the result is:

$$\ddot{\theta} = u - \theta - 254 - (206 - 2\theta - (232 - 3\theta)(\dot{\theta} - \theta)u^2 - 121u(-267 - \theta))u \tag{17}$$

When autovalidating this model, i.e. simply filling in the datapoints used during the run, the result is as shown in figure 4.13. The fitness function in the GP run unfortunately only evaluates the error in the output. It does not take into account the stability of the model. As a simple straightforward function, equation (17) maps the inputs fairly good to the output, but when simulated in Simulink the model is unstable and doesn't provide an answer.

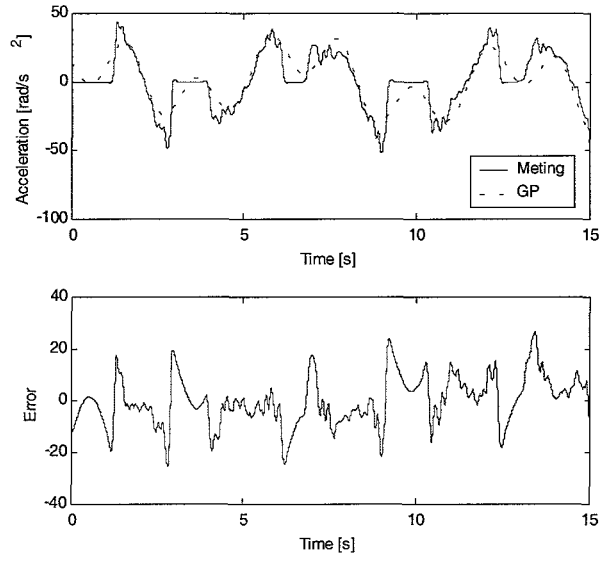

Figure 4.13: Result of first run

Second run

In the second run the following function set was used:

$$F = \{+, -, *, /, \exp\}$$

$$\ddot{\theta} = 2u\theta + 55u^2(2865456u^3 + 67u\dot{\theta} + 7035u + \frac{67}{648}\frac{\dot{\theta}^2}{u} - \frac{871}{36} - 67\dot{\theta}) \tag{18}$$

Again, after 164474 [s], the function is a map of input to output (as shown in figure 4.14), but as a model it is worthless as it is unstable. This model is not as good as the model of the first run. This is probably due to less computing time. The first run has run for twice as much time as this run. The problem was

that this run was stopped prematurely because the computer crashed and had to be restarted. There was no more time to run the test again.

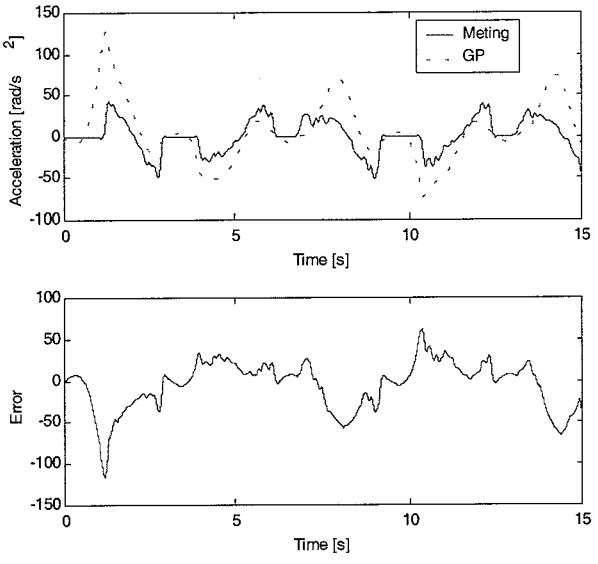

Figure 4.14: Result of second run

Third run

During the last run the exponential function was replaced by the sigmoid function. This resulted is the following equation:

$$\ddot{\theta} = \sigma\left(\frac{\dot{\theta}}{19}\right) \cdot \left(\sigma\left(-\frac{(83+792u\dot{\theta})}{4470048u^2}\right) + \frac{15u}{\sigma\left(\sigma\left(u\right)\right)}\right) \cdot \left(67 + \frac{\dot{\theta}}{\sigma\left(u\right)} + \frac{14}{\sigma\left(\sigma\left(67\dot{\theta}\right)\right)}\right) \tag{19}$$

This equation is very hard to comprehend in physical terms. This model is, just as the previous two, unstable. A plot of the function is shown in fig. 4.15.

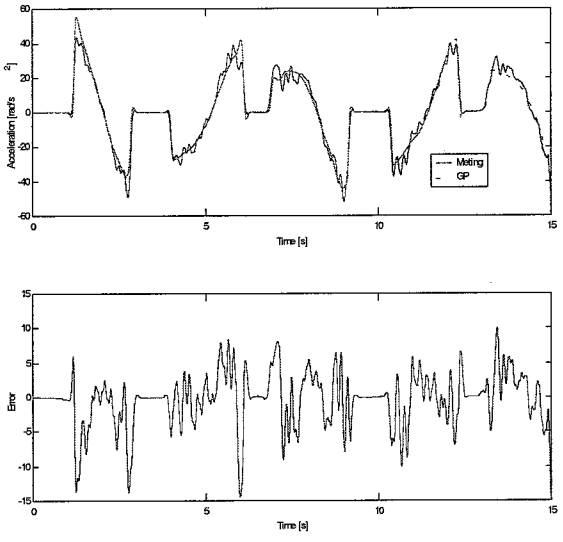

Figure 4.15: Result of third run

Chapter 5
Conclusion

In this report the basics of GP have been explained and illustrated with several problems. The simple problems were easily and exactly solved by the GP algorithm, whereas the bigger problems posed some problems. There are still many possibilities open to solve those problems. The GP algorithm can be adjusted. And as the fitness function has a major influence on how well GP performs, other fitness functions should be tried. More fitness cases could be used but as it took GP four days to come up with the solutions presented in paragraph 4.4 further improvements have not yet been tried for the pendulum problem.

Identification of a real system with GP did not result in a good model. The problem is that the stability of a model is not evaluated now, often resulting in mathematically good functions but unstable models. A recurring problem was a division by zero. The fitness function should be revised so that when a division by zero occurs a major penalty is given. To improve identification with GP differentiation and integration functions could be implemented in the GP program. It would then become unnecessary to differentiate or integrate data before hand and the order of a system could then be determined by GP.

By using the sigmoid function, prior knowledge of the system is put into the GP run. It is debatable wether this is acceptable. In an ideal situation (i.e. a very fast computer), no prior knowledge should be given to the GP run and the solution would be good all the same. Here, prior knowledge is used. The question is wether this prior knowledge will force the solution in a specific direction, thereby possibly missing important other solutions, or that it will merely help to speed up the convergence to a solution.

Human competitive results have already been achieved in several research fields. For example, controllers have been found by GP that are a significant improvement on controllers designed by humans [7]. Genetic Programming is becoming a powerful method for generating (mathematical) models for all sorts of problems, especially when taking into account the rapidly evolving speed of computer hardware. In the future it will be easier to compute complex models in less time, so the interest in GP will probably be growing in the coming years.

Bibliography

[1] A. Turing, *Mechanical Intelligence: Collected Works of A.M. Turing*, North-Holland, Amsterdam, 1948.

[2] J. Holland, *Adaptation in Natural and Artificial Systems : An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, MIT Press, Cambridge, 1975.

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

[4] W. Banzhaf, P. Nordin, R. Keller, and F. Francone, *Genetic Programming an Introduction on the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufman Publishers, Inc., 1998.

[5] J. Koza, D. Andre, F. Bennet III, and M. Keane, Use of Automatically Defined Functions and Architecture-Altering Operations in Automated Circuit Synthesis with Genetic Programming, in *Genetic Programming 1996: Proceedings of the First Annual Conference*, Cambridge, 1996, The MIT Press.

[6] J. Koza, Evolving the Architecture of a Multi-Part Program in Genetic Programming Using Achitecture-Altering Operations, in *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 695–717, Cambridge, 1995, MIT Press.

[7] J. Koza, M. Keane, J. Yu, F. Bennet III, and W. Mydlowec, Automatic Creation of Human Competitive Programs and Controllers by Means of Genetic Programming, genetic programming and evolvable machines **1**, 121–164 (2000).

[8] J. Koza, M. Keane, F. Bennet, J. Yu, W. Mydlowec, and O. Stiffelman, Automatic Creation of Both the Topology and Parameters for a Robust Controller by Means of Genetic Programming, in *Proceedings of the 1999 IEEE International Symposium on Intelligent Control/Intelligent Systms and Semiotic*, IEEE, 1999.

[9] G. Gray, T. Weinbrenner, D. Murray-Smith, L. Yun, and K. Sharman, Issues in Nonlinear Model Structure Identification Using Genetic Programming, in *Genetic Algorithms in Engineering Systems: Innovations and Applications*, IEEE, 1997.

[10] M. Willis, H. Hiden, P. Marenbach, B. McKay, and G. Montague, Genetic Programming: An Introduction and Survey of Applications, in *Genetic Algorithms in Engineering Systems: Innovations and Applications*, IEE, 1997.

[11] R. Barlow and A. Barnett, *Computing for Scientists: Principels of Programming with Fortran90 and C++*, Chichester : Wiley, 1998, 1998.

[12] B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

Appendix A
Prefix and infix notation

An example of a formula written in prefix notation is given in equation (20).

$$-+--76+543+21 \tag{20}$$

The equivalent expression in infix notation is given in equation (21).

$$(((7-6)-(5+4))+3)-(2+1) \tag{21}$$

The prefix notation is used because this notation is more convenient together with the tree form notation. To 'read' a tree start at the top and repeatedly take the left-most node until there are no more nodes. Go up one node and take the right-node. If there is no right-node, go up one node again. Again (if there is a left node) repeatedly take the left-most node until there are no more nodes. Repeat this process till all nodes are traversed. In the process of doing this write down the content of every node that is traversed for the first time. When finished the formula written down from the tree is a formula in prefix notation. See figure A.1 and follow the arrows for correct tree traversing.

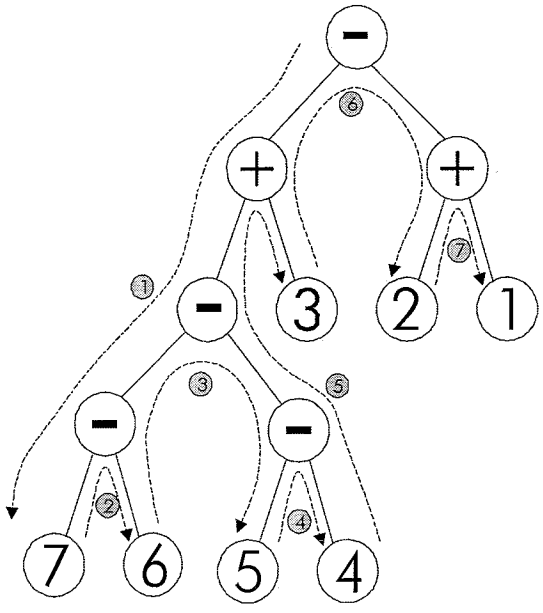

Figure A.1: Example of tree traversing