Searching for stability

Finding Lyapunov functions using GP

Master's Thesis

Iris Soute

DCT 2001-39
August 2001

TU/e

Coaching:

dr. ir. M.J.G. van de Molengraft

Eindhoven University of Technology
Faculty of Mechanical Engineering
Dynamics and Control Technology

Samenvatting

In dit verslag wordt uitgezocht of en hoe Genetisch Programmeren (GP) gebruikt kan worden om Lyapunov Functies te vinden.

Genetisch Programmeren valt onder het zogenaamde Evolutionary Computing (EC). GP zoekt naar oplossingen van problemen, gebruik makend van de evolutie theorie.

Lyapunov's theorie kan stabiliteit van systemen aantonen. Een groot voordeel van de Lyapunov theorie is dat het toepasbaar is op allerlei soorten systemen. Het grote nadeel echter is dat het erg moeilijk is om een Lyapunov function te vinden. Er bestaat namelijk geen standaard methode voor.

Zowel GP als de Lyapunov theorie zullen in dit verslag toegelicht worden. Vervolgens wordt er een algoritme ontwikkeld dat gebruik maakt van GP om op zoek te gaan naar de Lyapunov functies. Dit algoritme is geprogrammeerd in C++. Om het toegankelijker te maken voor een grotere groep van gebruikers, is er een grafische schil ontworpen in Matlab. Hierin kunnen makkelijke parameters ingesteld worden en kunnen achteraf de gevonden functies geëvalueerd worden.

Om het algoritme te testen is het vergeleken met verschillende andere methodes om Lyapunov functies te vinden. Kort samengevat zijn al deze methodes gebaseerd op het kiezen van een bepaalde vorm van de Lyapunov functie om vervolgens parameters te bepalen. Door de vorm van de functie vast te leggen, beperkt men zich tot een bepaalde klasse van resultaten.
Hierin verschilt GP: er hoeft van te voren geen vorm voor de functie vastgelegd te worden. Het algoritme bepaalt zelf de optimale vorm en parameters.
Er zal worden aangetoond dat GP in de meeste gevallen vergelijkbare of zelfs betere resultaten boekt dan de andere methodes.

Concluderend kunnen we zeggen dat GP een interessante manier biedt om Lyapunov functies te vinden. Verder is de verwachting dat GP een veelbelovende techniek biedt voor het oplossen van complexen problemen in het algemeen.

Abstract

In this report we will investigate if Genetic Programming (GP) can be used in the search for Lyapunov functions.

Genetic Programming is an Evolutionary Computing (EC) technique. It computes and evolves solutions, imitating the evolutionary process.

Lyapunov theory can prove stability of systems. An advantage of this theory is that it is applicable to all kinds of systems. A major drawback is that there exists no universal method for finding a Lyapunov function.

Both GP and Lyapunov will be elaborated in this report. Subsequently an algorithm based on GP to find Lyapunov functions is proposed. The algorithm is implemented in a C++ program and a graphical user interface is created in Matlab to enhance usability.

To test the algorithm, it is compared to several other techniques for finding Lyapunov functions. Basically, these methods all assume a fixed form for the Lyapunov function. This is where GP differs. One does not need to fix the form of the Lyapunov function beforehand. The GP will evolve the form of the Lyapunov function itself, thereby being able to access a more diverse set of solutions than the other techniques.
We will show that GP often leads to equivalent or better results compared with other techniques.

Concluding, we can say that GP offers an interesting technique for finding Lyapunov functions. Moreover, placing these results in a broader perspective, we expect that GP can be used for other complex problems in general.

Contents

Chapter 1

Introduction

In 1892 Lyapunov introduced a way to prove stability of mechanical nonlinear systems [1]. Lyapunov's theory was based on energy considerations. If a system, linear or nonlinear, is always dissipating energy, except at the origin, then the system must eventually settle down at the equilibrium point in the origin. So the energy must be a positive definite function for all non-zero states. This theorem can easily be extended to arbitrary nonlinear systems. Faced with a set of nonlinear differential equations, the basic procedure of Lyapunov's direct method is to generate a scalar 'energy-like' function for the dynamic system and examine the variation in time along the system's vector field.

The problem lies in finding such a Lyapunov function. The inability to find a Lyapunov function does not mean that the system is unstable. Several techniques, e.g. the variable gradient method [2] and Krasovskii's theorem [3] have been developed to find a Lyapunov function, but there is still no universally 'best' method for finding a Lyapunov function. The problem with these techniques is that before finding the Lyapunov function restrictive assumptions on the system have to be made.

The goal of this final project is to find out if Genetic Programming, an Artificial Intelligence technique, can be used in search of a Lyapunov function, as they are very hard to find. If a Lyapunov function is found, conclusions can be drawn concerning the stability of (non) linear systems, and also stable controllers for these systems can be built based on the Lyapunov function.

In Chapter 2 and 3 the some background Genetic Programming and Lyapunov's theory will be given. Chapter 4 will propose a method for using Genetic Programming to find a Lyapunov function. This method is implemented in a program, the Lyapunov Function Finder (LyFF), that is described in Chapter 5. Subsequently, LyFF will be tested on several problems in Chapter 6. Finally some conclusions and recommendations based on the tests will be posed in Chapter 7. In the last appendix the article [4], as published and presented at the GECCO2001-conference, is included.

Chapter 2

Genetic Programming - The Basics

Genetic programming (GP) is the technique for finding solutions to problems by imitating processes as seen in nature during the evolutionary process. Therefore, GP is considered a form of Artificial Intelligence (AI), or Evolutionary Computing (EC). These techniques are gradually gaining more interest in all sorts of research fields. 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 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 represents a potential solution to the problem.

2.1 Representation

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

$$y = \frac{a-b}{3} \tag{2.1}$$

can be presented as in Figure 2.1.

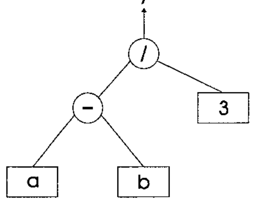


Figure 2.1: Tree-representation of Eq. (2.1).

The smallest part of a tree is called a node, connected nodes are called a branch. 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 2.1 has depth 3. The tree form is often used to display formulas in GP because it facilitates the use of genetic operations, as will become more clear later.

2.2 Functions and Terminals

The terminal and function sets are the alphabet of the programs to be made. They can be seen as LEGO-blocks that GP can freely use to build solutions.

2.2.1 The Terminal Set

The terminal set consists of inputs to the GP program and constants supplied to the GP program. They are called terminals because they terminate or end a branch of a tree in a tree-based GP. Each feature (input) of a problem becomes a part of the terminal set in a GP system. Thus, the features of the learning domain are just one of the blocks GP uses to build program structures. The features are not represented in any fixed way or in any particular place. In fact, the GP system can ignore an input altogether.

2.2.2 The Function Set

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

Of course, the user is free to program his or her own functions.

2.2.3 Choosing the Function and Terminal Set

The functions and terminals chosen for a GP run, should be powerful enough to be able to solve the problem at hand. For example, a function set that only contains the addition operator is not likely to solve complex problems. Alternatively, 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.

An 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, for example a very large number or zero.

A final piece of advice when choosing the function and terminal set, is not to spend too much time in designing dedicated functions or terminals. Experience has pointed out the GP is very creative in taking simple functions and terminals and combining them to fulfill its needs. In fact, GP is known to ignore complex functions in favor of the simple functions during a run. If GP is not able to find a solution with the simple set, it is time to start designing your own functions and terminals.

2.3 Fitness and Selection

GP is imitating the evolutionary process. According to Darwin's evolution theory, individuals that are more 'fit' than others will survive in natural selection and will reproduce, to create (hopefully) even fitter new individuals. This is exactly what is being simulated by GP: It starts by creating a group of solutions, a population, and checks how 'fit' they are to solve a problem. Subsequently the fittest individuals are selected to reproduce. This process is repeated with the offspring over many generations until a satisfying individual is found. In this sense, the algorithm is more similar to breeding than to natural selection.
The process of selecting the best individuals is simply called *selection*. The process of checking how fit an individual is done by the *fitness function*.

2.3.1 The Fitness Function

As the fitness function has great effect on the evolution it is the most difficult and most important concept of GP. Therefore we need to design the fitness function very carefully. It needs to give a graded 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. This is accomplished by assigning each individual a numeric value, the fitness value, that is based on criteria set by the user and that corresponds to the appropriateness of a solution.

Fitness is usually computed over a training set that is composed of a number of fitness cases. The number of fitness cases should be sufficiently large as to produce a range of different numerical 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.

A little example to illustrate the above: we would like GP to evolve an individual that learned the patterns in Table 2.1, that is, a program that could predict the output column by knowing only the value in the input column. Obviously, this example is very simple and a program representing the function $f(x) = x^2 + x$ is a perfect match.

Table 2.1: Input and output values for a training set.

	Input	Output
fitness case 1	1	2
fitness case 2	2	6
fitness case 3	4	20
fitness case 4	7	56
fitness case 5	9	90

One simple fitness function that we could use for this problem would be to calculate the sum of the absolute difference between the actual output of the program and the output given by the training set, thus the error. For the training set given in Table 2.1 the fitness function would then formally be:

$$f_p = \sum_{i=1}^{5} |g_i - o_i| \tag{2.2}$$

with:

f_p	fitness value
g_i	the output from a GP program p on the ith example of the training set
o_i	the output of the ith example of the training set

As p_i gets a closer to o_i, the fitness gets better. A perfect solution would have fitness zero.

2.3.2 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. An individual is given a probability (to produce offspring) of

$$p_i = \frac{f_i}{\sum_i f_i} \tag{2.3}$$

  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.

2.4 Genetic operators

The first population of a run usually has very low fitness. During evolution the initial population is transformed by the use of genetic operators. While there are many operators the principal GP genetic operators are crossover, mutation and reproduction. Their implementations are based on phenomena seen in nature: crossover is equivalent to what is called 'sexual recombination', or the mating between two parents, where genes get mixed, resulting in a child bearing features of both of its parents. The mutation operator is analog to mutation of genes or cell by exogenous influences, e.g. the mutation of cells under influence of sunlight. The reproduction operator matches asexual reproduction of, for example, single-celled organisms.

During a GP run, first an individual is selected from the population by means of a selection method described in Section 2.3. Then the operation that will be performed on the individual is selected. All operations have an assigned probability value and this value corresponds to the probability that that operation will be selected.

2.4.1 Crossover

The most important and most used 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 2.3. The crossover operation combines the properties of two parents by swapping a part of one parent with a part of the other, see Figure 2.2. The crossover point is randomly chosen.

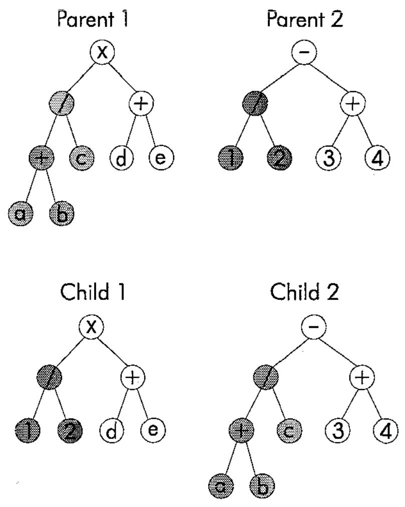


Figure 2.2: Cross-over

Crossover has an drastic effect on the offspring. While mutation and reproduction operators, as we will see later, result in offspring that differ only slightly from the parents, crossover can generate offspring that is completely different from the parents. This feature is needed during evolution to make great leaps forward, as for example two mediocre parents can create a near perfect solution. The downside of this is that great leaps backwards also occur, and unfortunately they occur more frequently then their counterparts: as much as 75% of the crossover operations is lethal to the offspring, in the sense that their fitness value is considerably lower than that of their parents. Much research is done on the exact influence of the crossover operator on the fitness, which might lead to better understanding and improvements of the operator [5].

2.4.2 Mutation

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

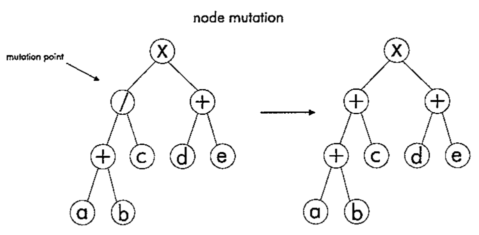
1. a function can only replace a function or a terminal can only replace a terminal.

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 here, see Figure 2.3.

2.4.3 Reproduction

The reproduction (also called copying) operator selects an individual from the population based on its fitness value. Then the individual is copied and placed back in the population, resulting in two versions of the same individual in the population.



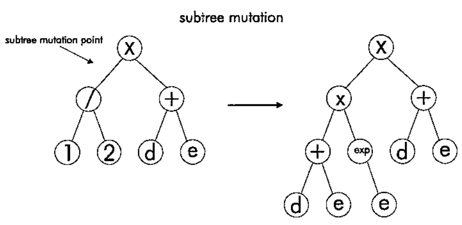


Figure 2.3: Mutation

2.5 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. 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 a run. These parameters are often expressed in weighted values. A crossover probability of 0.7, a mutation probability of 0.3, and a reproduction probability of 0.1 are often used values in GP-runs.
- Selection method. See Section 2.3.
- Maximum individual size. This value refers to the maximum depth the individuals can obtain. When choosing 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.
- Creation type at initialization. When starting up a run, the population is filled with individuals that are randomly created. This setting specifies the depth at which the individuals should be created. We can choose whether all individuals should reach full depth, or that the depth should be variable, or any other user defined setup. This setting influences the speed at which the first population is built up and also has some influence on the diversity of the population.

2.6 A Basic GP Algorithm

Now all separate parts have been discussed, it is time to combine them in an overall GP algorithm for a basic GP run. There are two ways to execute a run, a generational approach and a steady-state approach. However, first we will review the preparatory steps before getting a GP to run.

2.6.1 Summary of the preparatory steps

Here are the preliminary steps in a GP run, which we have already described in detail in this chapter:

1. Define the terminal set
2. Define the function set
3. Define the fitness function and the training set
4. Define parameters such as population size, crossover probability, and termination criterion.

Once these steps are completed the run can begin. How it proceeds depends on whether it is generational or steady state.

2.6.2 Generational versus steady state GP algorithm

There are basically 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 one of the selection methods described in Section 2.3. From an old generation individuals are selected to create new individuals by crossover, mutation, or reproduction until a new generation is filled with newly created individuals. The old generation is then discarded, and the process is started over again to produce a new generation until the termination criterion is fulfilled, for example if a perfect solution has been found or if a certain number of generations has passed.

The other option is steady state GP. No new generations are created now. The GP run starts to create a population of individuals. Then, tournament selection is applied. As described, not all individuals are allowed to compete, just a small set of them taken randomly from the population. The fitness is determined of the individuals in the tournament, and subsequently the winner or winners are selected using the selection algorithm. The genetic operators are applied to the winner or winners and instead of putting the newly created individuals in a new generation, they are put back in the existing population, replacing the individuals that 'lost' during the tournament. Note that the 'winning' individuals are also returned to the population. Especially for large problems steady state GP can reduce computing time, because of the use of tournament selection but also because of the fact that only one generation has to be maintained in the computer memory instead of two. But there is no noticeably difference in the results when using either method.

Chapter 3

Lyapunov Theory

Stability is important for many applications. Unstable systems can exhibit unwanted and sometimes destructive behavior. Therefore stability has to be analyzed. For linear systems several methods are available: analysis of the system in time domain or in frequency domain should provide the researcher with enough information about the stability of the system. For nonlinear systems matters are somewhat different. The techniques used for analysis of linear systems are not applicable here, since direct solution of nonlinear differential equations is generally impossible, and frequency domain transformations do not apply. Unfortunately, there is no universal technique for analysis of nonlinear systems. Many different methods have been proposed, Lyapunov's theory being one of them [1],[6].

3.1 Concepts of Stability

Throughout this report we will only be dealing with systems that are autonomous. These systems are time-invariant and can be represented as follows:

$$\dot{\mathbf{x}} = f(\mathbf{x}) \tag{3.1}$$

A point $\mathbf{x} = \mathbf{x}^*$ is called an equilibrium point of the system if once $\mathbf{x}$ is equal to $\mathbf{x}^*$, it remains at $\mathbf{x}^*$ for all future time. For autonomous systems of the form (3.1) the equilibrium points can be determined by finding the real roots of Eq. (3.2).

$$f(\mathbf{x}) = \mathbf{0} \tag{3.2}$$

If a system starts near an equilibrium point and it stays near this point as time goes to infinity, the equilibrium point is called stable. If it moves away from the equilibrium point it is called unstable. If an equilibrium point is stable and, in addition, the system tends towards the equilibrium point as time goes to infinity, the equilibrium point is called asymptotically stable.

3.2 Lyapunov's Direct Method

Lyapunov's direct method is a generalization of the energy concepts associated with a mechanical system: the motion of a mechanical system is stable if its total energy decreases all the time. For mechanical systems we can formulate an energy function $E(\mathbf{x})$. This is however not possible for systems that have no physical meaning and are expressed in a mathematical form. So Lyapunov came up with the idea to construct a scalar *energy-like* function, the Lyapunov function ($V(\mathbf{x})$), for a system. If this Lyapunov function is always positive definite and decreases in time (this can be investigated by computing the derivative of the Lyapunov function), the systems equilibrium point is stable.

The power of this method comes from its generality: it is applicable to all kinds of systems, whether they are time-varying or time-invariant, finite dimensional or infinite dimensional. Furthermore, this method allows stability to be investigated without the need to simulate. The limitation of the method lies in the fact that it is often difficult to find a Lyapunov function for a given system. Since there is no generally effective approach for finding Lyapunov functions, one has to use trial-and-error, experience or intuition to search for appropriate functions.
Note that the Lyapunov theorems are *sufficiency* theorems. If for a particular choice of Lyapunov function candidate the conditions are not met, a conclusion on the stability or instability of the system cannot be drawn - the only conclusion is that a different Lyapunov function candidate should be tried.

We can define a Lyapunov function as follows:

Definition 3.1 *If, in a ball $\mathbf{B}_{R_0}$, the function $V(\mathbf{x})$ is positive definite and has continuous partial derivatives, and if its time derivative along any state trajectory of the system is negative semi-definite, i.e. $\dot{V}(\boldsymbol{x}) \leq 0$ then $V(\mathbf{x})$ is said to be a Lyapunov function for the system.*

A function is said to be *positive definite* if $V(\mathbf{0}) = 0$ and if $\mathbf{x} \neq \mathbf{0} \Rightarrow V(\mathbf{x}) > 0$. A function $V(\mathbf{x})$ is *negative definite* if $-V(\mathbf{x})$ is positive definite. $V(\mathbf{x})$ is *positive semi-definite* if $V(\mathbf{0}) = 0$ and $V(\mathbf{x}) \geq 0$ for $\mathbf{x} \neq \mathbf{0}$; $V(\mathbf{x})$ is *negative semi-definite* if $-V(\mathbf{0})$ is positive semi-definite. The prefix "semi" is used to reflect the possibility of V being equal to zero for $\mathbf{x} \neq \mathbf{0}$ [6].

3.3 Lyapunov Theorems for Stability

Lyapunov stated two theorems for the stability of equilibrium points, a local stability theorem and a global stability theorem. Starting with the local stability theorem:

Theorem 3.1 (Local Stability) *If, in a ball $\mathbf{B}_{R_0}$, there exist a scalar function $V(\mathbf{x})$ with continuous first partial derivatives such that*

- *$V(\mathbf{x})$ is positive definite (locally in $\mathbf{B}_{R_0}$)*
- *$\dot{V}(\boldsymbol{x})$ is negative semi-definite (locally in $\mathbf{B}_{R_0}$)*

then the equilibrium point $\boldsymbol{0}$ is stable. If, actually, the derivative $\dot{V}(\boldsymbol{x})$ is locally negative definite in $\mathbf{B}_{R_0}$, then the stability is asymptotic.

Assuming that $V(\mathbf{x})$ is differentiable, the derivative with respect to time can be found using the chain rule:

$$\dot{V} = \frac{dV(\mathbf{x})}{dt} = \frac{\partial V}{\partial \mathbf{x}}\dot{\mathbf{x}} = \frac{\partial V}{\partial \mathbf{x}}\mathbf{f}(\mathbf{x}) \tag{3.3}$$

The theorem for local stability can be extended to a theorem for global stability. Therefore the ball $\mathbf{B}_{R_0}$ needs to be extended to the whole state-space. On top of that we need to make sure that the contour curves of $V(\mathbf{x})$ remain to be closed curves: $V(\mathbf{x})$ must be radially unbounded, meaning that $V(\mathbf{x}) \to \infty$ as $\|\mathbf{x}\| \to \infty$ (in other words, as $\mathbf{x}$ tends to infinity in any direction). If the curves are not closed, it is possible for the system trajectories to drift away from the equilibrium point, even though the state keeps going through contours corresponding to smaller and smaller V_α's, as shown in Figure 3.1.
Keeping this in mind, the theorem for global stability is as follows:

Theorem 3.2 (Global Stability) *Assume that there exists a scalar function V of the state $\boldsymbol{x}$, with continuous first order derivatives such that*

- *$V(\mathbf{x})$ is positive definite*
- *$\dot{V}(\boldsymbol{x})$ is negative semi-definite*

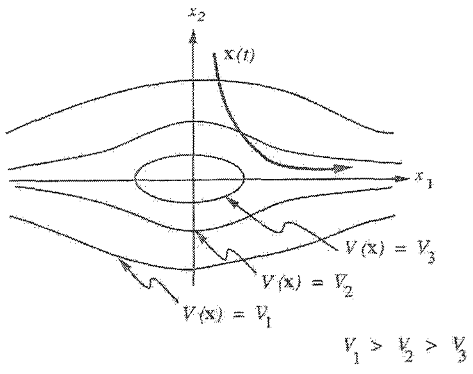


Figure 3.1: Motivation of the radial unboundedness condition

- $V(\mathbf{x}) \to \infty$ *as* $\|\boldsymbol{x}\| \to \infty$

then the equilibrium at the origin is globally stable. If $\dot{V}(\boldsymbol{x})$ *is negative definite, then the origin is globally asymptotically stable.*

3.4 The Region of Attraction

An important quality of an equilibrium point is its region of attraction, or, how far away can we start from an equilibrium point and still be able to end up in that very same equilibrium point?

Theorem 3.3 *Consider the autonomous equation*

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}), f(\boldsymbol{0}) = \boldsymbol{0} \tag{3.4}$$

Let $V(\mathbf{x})$ *be a scalar function. Let* Ω *designate a region where* $V(\mathbf{x}) > 0$. *Assume that* Ω *is bounded and that within*

- $V(\mathbf{x}) > 0$
- $\dot{V}(\boldsymbol{x}) < 0$

Then the origin is asymptotically stable and all motions starting in Ω *converge to the origin as* $t \to \infty$. *The region* Ω *is called a region of attraction.*

Note that the region obtained by the inequality given above depends on the choice of $V(\mathbf{x})$ and different choices of $V(\mathbf{x})$ may yield different estimates of the region of attraction. Then the union of these regions is contained in the exact region of attraction. Thus, improved estimates can be obtained by taking different Lyapunov functions.

3.5 Convergence or The Decay Rate

Not only do we need to know that a system will end up in an equilibrium point, sometimes it is also important to predict how fast it will get there. Therefore, we need to ascertain that the solution will converge exponentially to its equilibrium.

If $V(\mathbf{x})$ satisfies the inequality

$$\dot{V}(\mathbf{x}) \leq -\alpha V(\mathbf{x}) \tag{3.5}$$

for some constant $\alpha > 0$, then

$$V(t) \leq V(0)e^{-\alpha t} \tag{3.6}$$

So, if $V(\mathbf{x})$ is a non-negative function, Eq. (3.5) guarantees the exponential convergence of $V(\mathbf{x})$ to its equilibrium point.

Both the theory for GP and Lyapunov have now been surveyed. In the next chapter, the two are put together to create the algorithm that will search for Lyapunov function candidates.

Chapter 4

The GP Algorithm for finding Lyapunov Functions

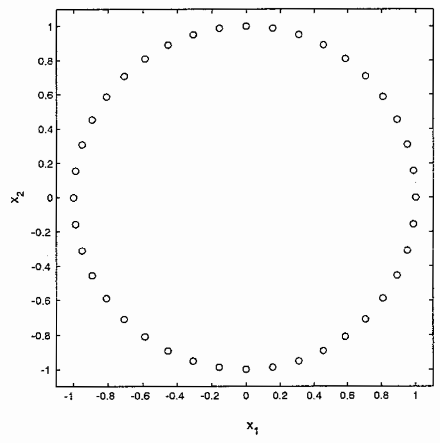
The fitness function is the most important function during a GP run. The function will assign a fitness value to all the solutions and can thereby discriminate between the appropriateness of solutions. We want the run to converge to the best solution, and therefore it is very important to set-up a good fitness function. Also, we have to define on what set of the domain the fitness function must be applied, the so called fitness cases.

4.1 Fitness Cases

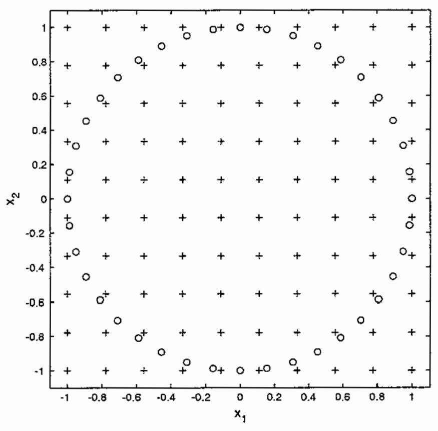
Let space X^0, where $X^0 \subset \mathcal{R}^2$, be a compact and connected region of the state space, such that the origin is included in X^0. To prove local stability, we need to find a Lyapunov function that guarantees $V(\mathbf{x}) > 0$ and $\dot{V}(\mathbf{x}) < 0$ within X^0. Therefore, X^0 is discretized into a finite set of points (fitness cases), to be used during the GP run. The fitness cases are divided into two grids: a Coarse Grid and a Fine Grid, see Figure 4.1. The Coarse Grid is always calculated such that it includes the points on the axes in state space to assure a positive definite function containing terms in all states. It encompasses the points on the edge of the grid, and can for example be in rectangular or circular form. The origin is excluded from the Fine Grid because, as we will see later, we will be testing if $V(\mathbf{x}) > 0$ and $\dot{V}(\mathbf{x}) < 0$. In the origin $V(\mathbf{x}) = 0$ and $\dot{V}(\mathbf{x}) = 0$. For a Lyapunov function this is correct, but it would not fulfill the testing inequalities stated above, thus unintentionally introducing errors. Therefore the origin is omitted.
The reason for splitting up the fitness cases in two grids will become in Section 4.2.1.

4.2 The Algorithm

Now the fitness cases have been defined, we need a function that will test the individuals for all these fitness cases, and subsequently assigns them a numerical value. The function is split up in several parts: the most important one being the one that checks if the individual fulfills the requirements for a Lyapunov function. The other parts of the function include parts that try to maximize the region of attraction, maximize the decay rate, or try to enforce closed contour curves. The user can choose which of these three last part(s) are to be used. He can for example choose to only maximize the region of attraction, or try to maximize both the region of attraction and the decay rate.
In essence, all parts work in the same way. For every part conditions have been drawn up to achieve its objective (maximal region of attraction, etc.). An individual is then tested on every point on the grid (the fitness cases) to check if the conditions hold. If a condition holds, nothing happens. If it does not hold, the individual will get one point added to its fitness value. The



(a) Coarse Grid



(b) Coarse and Fine Grid

Figure 4.1: Fitness cases in grids

total fitness value of an individual is the sum of the fitness values of the separate parts of the fitness evaluation.
As all individuals start with fitness zero, the individuals that end up with fitness zero fulfill all conditions. Individuals that have a low fitness value did perform quite well, but missed on a few fitness cases. Consequently, individuals that have a high fitness value performed badly. As GP will try to achieve fitness zero, convergence to an optimal solution will find place.

4.2.1 Finding a Lyapunov Function

To generate Lyapunov function candidates all individuals are tested on the fitness cases to check if $\dot{V}(\mathbf{x}) < 0$ and $V(\mathbf{x}) > 0$. $V(\mathbf{x})$ can easily be tested by evaluating the function on every grid point. This results in a numerical value per fitness cases. For every fitness case where $V(\mathbf{x}) < 0$ a penalty is given to the fitness value. The same goes for the evaluation of $\dot{V}(\mathbf{x})$. The value of $\dot{V}(\mathbf{x})$is approximated numerically. For this purpose we use a 3-point central difference derivation approximation. The calculation of $\dot{V}(\mathbf{x})$ is as follows (for a two dimensional system):

$$\dot{V}(\mathbf{x}) = \frac{V(x_1+\Delta x, x_2) - V(x_1-\Delta x, x_2)}{2\Delta x} f(x_1) + \frac{V(x_1, x_2+\Delta x) - V(x_1, x_2-\Delta x)}{2\Delta x} f(x_2) \tag{4.1}$$

where Δx is chosen small: $\Delta x = 10^{-6}$.
This way all fitness cases can be tested for the inequalities $V(\mathbf{x})> 0$ and $\dot{V}(\mathbf{x})< 0$, which will result in a fitness value for this individual. Step by step the fitness function for finding a Lyapunov candidate is then:

1. For $i = 0 \dots N$ calculate $V(\mathbf{x}_i)$. If $V(\mathbf{x}_i) < 0$ then $f_l = f_l + 1$.
2. For $i = 0 \dots N$ calculate $\dot{V}(\mathbf{x}_i)$. If $\dot{V}(\mathbf{x}_i) > 0$ then $f_l = f_l + 1$.

where:

f_l	fitness value assigned to an individual, due to the Lyapunov evaluation
N	number of points on the grid
$\mathbf{x}_i$	the ith fitness case on the grid
$V(\mathbf{x}_i)$	value of $V(\mathbf{x})$ evaluated on $\mathbf{x}_i$
$\dot{V}(\mathbf{x}_i)$	value of $\dot{V}(\mathbf{x})$ evaluated on $\mathbf{x}_i$

For the sake of speed, the individuals are first tested this way on the Coarse Grid. This provides a fast sifting of solutions. As the run is started the individuals are created randomly. In the first generation the fitness value of the individuals will be high, meaning that they are nowhere near being a Lyapunov function candidate. So sifting them out before testing them on the Fine Grid, which is time consuming because of the larger number of fitness cases, would mean computing time reduction. However, rejection does *not* mean that those individuals are thrown out of the population! It just means that they are not evaluated on the Fine Grid. We raise the fitness value of those individuals by the number of fitness cases on the Fine Grid, and skip further evaluation, thereby gaining speed. These individuals will receive a large fitness value, and will have less chance to create offspring.
In the end, the function with the lowest fitness value (i.e. the one that received the least penalties) is the best Lyapunov candidate.

4.2.2 Closing the Curves

As explained before, to prove stability we would like the contour curves of the Lyapunov function to be closed. To promote closed curves during a run, the following is proposed:

1. Calculate the minimal (V_{min}) and maximal (V_{max}) value of $V(\mathbf{x})$ on the Coarse Grid
2. Calculate the difference between the maximum and minimum value of $V(\mathbf{x})$: $\Delta V = V_{max} - V_{min}$
3. Punish the individuals that have a larger ΔV than some predefined value C, e.g. if $\Delta V > C$ then $f_c = 10(\Delta V)^2$

4.2.3 Maximizing the Region of Attraction

To maximize the region of attraction we have to find the largest closed curve within the set X^0. A way to do this is to determine the minimal value of $V(\mathbf{x})$ on the edge of the subset (which is represented by the Coarse Grid). If a minimal value V_{min} is found then we assume that within the subset there exists a closed curve $V(\mathbf{x}) = V_{min}$. Now the region of attraction X is estimated by

$$X = \{x \in X^0 \mid V(\mathbf{x}) \leq V_{min}\} \tag{4.2}$$

It is now very easy to check whether a point on the Fine Grid is within the region $V(\mathbf{x}) \leq V_{min}$, or not. So the fitness value is determined as follows:

1. For $i = 0 \ldots N$. If $V(\mathbf{x}_i) > V_{min}$ then $f_r = f_r + 1$.

So, during the evaluation of the Fine Grid an individual will receive punishment for points that lie outside the region of attraction. The more points lie outside the region, the more an individual will be punished. Because an individual with fitness zero is regarded as 'best' by the GP algorithm, the algorithm shall now try to evolve individuals that have as much points as possible within the region of attraction, thereby maximizing the region of attraction.

There are situations imaginable where V_{min} on the Coarse Grid does not correspond with a closed curve, or at least not with the closed curve that corresponds to the largest region of attraction, in which the method described above would fail.
Although not implemented, a better way would be a function that evaluated on expanding circles:

1. Start with a very grid that is a very small circle around the origin.
2. Evaluate the Lyapunov conditions.
3. Enlarge the radius of the circle a little.

4. Repeat step 2 and 3 while the Lyapunov conditions hold.

5. Stop as soon as $\dot{V}(\mathbf{x}) = 0$ is encountered. Now calculate V_{min} on the current circle. This V_{min} corresponds to the largest region of attraction for this individual.

To assign a numerical value to the size of the region of attraction Johansen [7] defines it as the largest square with radius r: $[-r, r] \times [-r, r] \subset X^0$ within the region of attraction. We will adopt this definition here, because it will simplify comparisons. After a run r is calculated using Matlab.

4.2.4 Maximizing the Decay Rate

To maximize the decay rate α the function proceeds as follows:

1. Start with choosing a small value for α
2. If, for $i = 1 \ldots N$, $\dot{V}(\mathbf{x}_i) > -\alpha V(\mathbf{x}_i)$ then $f_d = f_d + 1$
3. If $f_d = 0$ then raise α, e.g. $\alpha = 1.25 \times \alpha$
4. Repeat steps 2 and 3 until a pre-defined number of generations has passed.

As soon as $f_d = 0$ is encountered the algorithm will save that particular individual and its decay rate. So, in this case, the best individual of a run will not be the individual with the lowest fitness value at the end of the run, but the individual that was saved at the largest decay rate.

4.2.5 Calculating the Total Fitness Value

To calculate the total fitness value of an individual, we have to sum the fitness values of the separate parts, or sum the fitness values of the parts we have chosen to evaluate. So:

$$f = f_l + f_c + f_r + f_d \tag{4.3}$$

4.3 The Effect of Discretization

By defining fitness cases in grids and evaluating $V(\mathbf{x})$ and $\dot{V}(\mathbf{x})$ on these grids, we can only draw conclusions about the appropriateness of the Lyapunov function candidates *on these points* during the run. Therefore, the candidates need to be tested afterwards (in Matlab for example)to check if they are Lyapunov functionfor X^0. Generally, the discretization has no influence on the results, as the GP generated Lyapunov function candidates afterwards check out to be Lyapunov functions. But sometimes problems arise due to the discretization.
One of the problems that arises during a run, is that for the fitness cases the solution might seem a Lyapunov function, but when checked afterwards it isn't, because it shows 'spots' where $\dot{V}(\mathbf{x}) > 0$. Also, because of the complexity of the solutions found by GP, it is often not possible to evaluate the solutions analytically. Both problems are addressed here.

4.3.1 During a Run

It can occur that GP deems a solution as the best solution for the problem and ends the run. Evaluation afterwards sometimes shows that in fact the solution is not a Lyapunov function, because it has 'spots' within X^0 where $\dot{V}(\mathbf{x}) > 0$. The reason that GP did not 'see' this, is that the spots are precisely positioned between the fitness cases, of which an example is shown in Figure 4.2. The solution to this is of course to use a finer grid. However, more fitness cases result in an increase of computing time. So, this is a bit of a trade-off between accuracy and computing time. This phenomenon is problem-specific, but as a rule of thumb it can be stated that the more complex the problem is, the more grid points are needed to avoid the phenomenon described above.

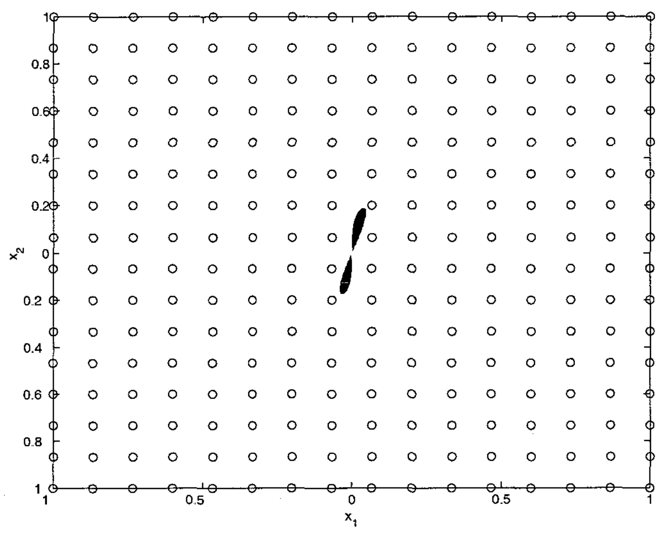


Figure 4.2: A typical problem that arises due to the discretization of the state space: 'spots' (indicated in black) where $\dot{V}(\mathbf{x}) > 0$

4.3.2 Evaluation Afterwards

If a GP run has found a Lyapunov function candidate, we are still not sure whether it is really a Lyapunov function. Ideally, we would check this analytically afterwards, but as problems become more complicated, the Lyapunov function candidates also get more difficult to evaluate analytically, and even Matlab (which uses the MAPLE-kernel) or Mathematica are not able to evaluate the functions analytically as the candidates get too complex. Therefore we have chosen to check the Lyapunov candidate numerically. The candidate is checked, just as during the run, on a grid, but now on a much more dense grid. In our case, the candidates are tested on grids of 100.000 points in X^0, as this was near to the maximal number of points that Matlab could process. In Section 4.3.3 a method for calculating the density of a grid is proposed to ascertain that the grid is sufficiently dense to be able to draw the right conclusions.

4.3.3 A Sufficiently Dense Grid

To ascertain whether a candidate is a Lyapunov function or not we need to test it for the Lyapunov criteria. Because of the complexity of the candidates, this is not always possible analytically. Therefore we need to create a sufficiently dense grid, on which we are going to test the candidate.

To guarantee that $V(\mathbf{x}) > 0$, for all $\mathbf{x}$ within the set $X^0 = [-1,1] \times [-1,1] \subset \mathcal{R}^2$, we need to make sure that within X^0

$$V_{min} - \max\left(\frac{\partial V(\mathbf{x})}{\partial \mathbf{x}}\right)\Delta r > 0 \tag{4.4}$$

where (see Fig. 4.3)

Δr	$=$	grid size
V_{min}	$=$	$\min(V(A), V(B), V(C), V(D))$
$\max\left(\frac{\partial V(\mathbf{x})}{\partial \mathbf{x}}\right)$	$=$	maximal derivative over the area encompassed by A, B, C and D

Starting with X^0 in its totality, we verify if the inequality in Eq. (4.4) holds. If not, the domain is subdivided into four segments, and each segment is checked. This process is continued until the inequality holds. In this way all segments in X^0 are to be checked and the smallest Δr found is the minimal distance between grid points needed.

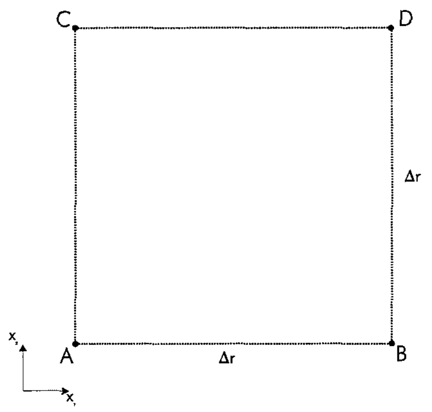


Figure 4.3: Grid segment

The algorithm is now outlined. In the next chapter we will continue to explain how the algorithm is actually implemented.

Chapter 5

The Application

The algorithm described in the previous chapter has been implemented in an existing program for GP calculations: The Genetic Programming Kernel, version 0.5.2. A description of the GP Kernel can be found in [8]. The implementation of the algorithm and the GP Kernel together form a new application: the Lyapunov Function Finder (LyFF).
To facilitate the use of LyFF, a Graphical User Interface (GUI) has been developed in Matlab. This GUI creates an configuration file in the Matlab MAT-format for easy configuration of LyFF. After startup of LyFF it will read this file and use the parameters as set in the file. During the run some values and results are written back to the file.
After the run the file can be read by Matlab to examine the results. Some Matlab-functions are available for this purpose, although as of yet they have not been implemented in the GUI.
To use LyFF, Sections 5.1, 5.2 and 5.3 should be read. More information on the implementation of the algorithm or the configuration file can be found in Sections 5.4 and 5.5, although to read these sections the reader is presumed to have some knowledge of C or C++. Development of applications is an on-going process, and LyFF is no exception. Therefore a ToDo-list and a known error list can be found in the last section of this chapter.

5.1 Setting up the application

5.1.1 Computer requirements

The speed at which the program will run is directly related to the processor speed and the size of the internal memory of the computer. The program will run on at least a 133MHz processor with 32Mb memory, but it will be slow. Most tests have been performed on a 500MHz Pentium 2 Pro processor with 128Mb internal memory. On this machine the tests ran at a sufficient pace.

To use the GUI Matlab 6 (release 12) is necessary, together with the Symbolic Toolbox. Since Matlab has changed its GUI format the GUI is unfortunately not backwards compatible with Matlab 5. However, the MAT-format has not been changed, so one can manually change the configuration file using any version of Matlab, and then run the program in a DOS box. For more information on the configuration file, see Section 5.5.

5.1.2 Setup

The installation file *setupex.exe* for both LyFF and the GUI can be found on the included disk. Running the setup will install the files on the computer. The user is prompted to enter the installation path. If Matlab is present, the directory that holds the GUI (in the installation path) must be added to the Matlab path by the user. To do this start Matlab 6 and press File| Set Path. Then add the GUI-directory to the path.

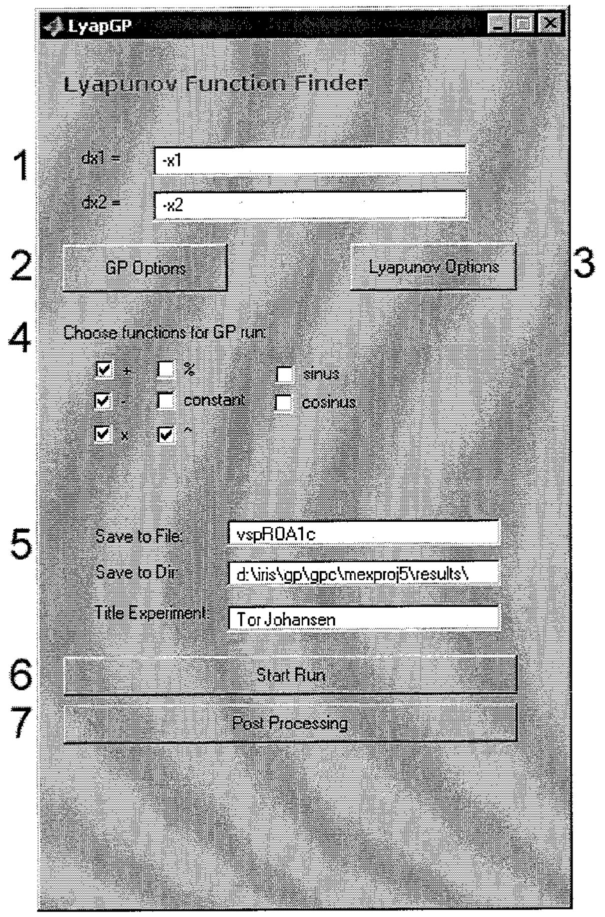
The program can be uninstalled using the uninstall option in the Add/Remove Programs in the Configuration Panel of Windows.

5.2 Using the Graphical User Interface

To run LyFF a configuration file is needed. As the configuration file is a standard MAT-file, the configuration file can be edited manually. But as there are many options, a GUI has been developed to facilitate and visualize the editing of the configuration file. All values entered in the GUI are automatically saved to the configuration file.

5.2.1 The Main Window

To start the GUI, start Matlab and navigate to the directory where you have installed LyFF.exe. Type LyFF. A window as shown in Figure 5.1 will pop-up.



dx1 = -x1

dx2 = -x2

Choose functions for GP run:

[✓] + [ ] % [ ] sinus

[✓] - [ ] constant [ ] cosinus

[✓] x [✓] ^

Save to File: vspR0A1c

Save to Dir: d:\iris\gp\gpc\mexproj5\results\

Title Experiment: Tor Johansen

Figure 5.1: Main window

To set up a run the following steps need to be taken, the numbers corresponding to the numbers in Figure 5.1:

1. Enter the system for which a Lyapunov function should be found. Remember that the variables should be in x_1 and x_2. At this time, the LyFF only works for two-dimensional systems.

2. Press this button to open a window to set GP options. This window will be explained in Section 5.2.2.

3. Press this button to open a window to set the search-domain and other options. This will be explained in Section 5.2.3.

4. Choose the functions you wish the GP algorithm to use during the run.

5. Enter the file name and directory (don't forget the backslash (\) at the end!) where output files should be named and written to. During the run three output files with information about the run are created. The 'Title Experiment' field is optional, it will be used in one of the output files. For more information about the output files, see Section 5.5.

6. If everything is set, push this button. A DOS-box will open and the run is started. If desired, the run can be ended manually by pressing any key. No information will be lost, as the output-files will still be written if you manually end the run. During a run you can minimize the DOS-box and proceed to use Matlab or any other program as you are used to, although the computer will be a little slower, due to the memory needed for the LyFF. However, when using Matlab, before moving on to the next step do not forget to change the working directory to the directory where you had put `mexproj5.exe`! This is still a minor inconvenience in the GUI. If the run ends automatically, the DOS-box will close by itself.

7. If the run has ended and the DOS-box is closed, press this button. Amongst other things, this will cause the derivative of the Lyapunov function (which was not calculated symbolically during the run) to be calculated and saved to the configuration file as a Matlab sym-object. In some cases, e.g. when the solution is very long, this might take a while.

5.2.2 GP Options Window

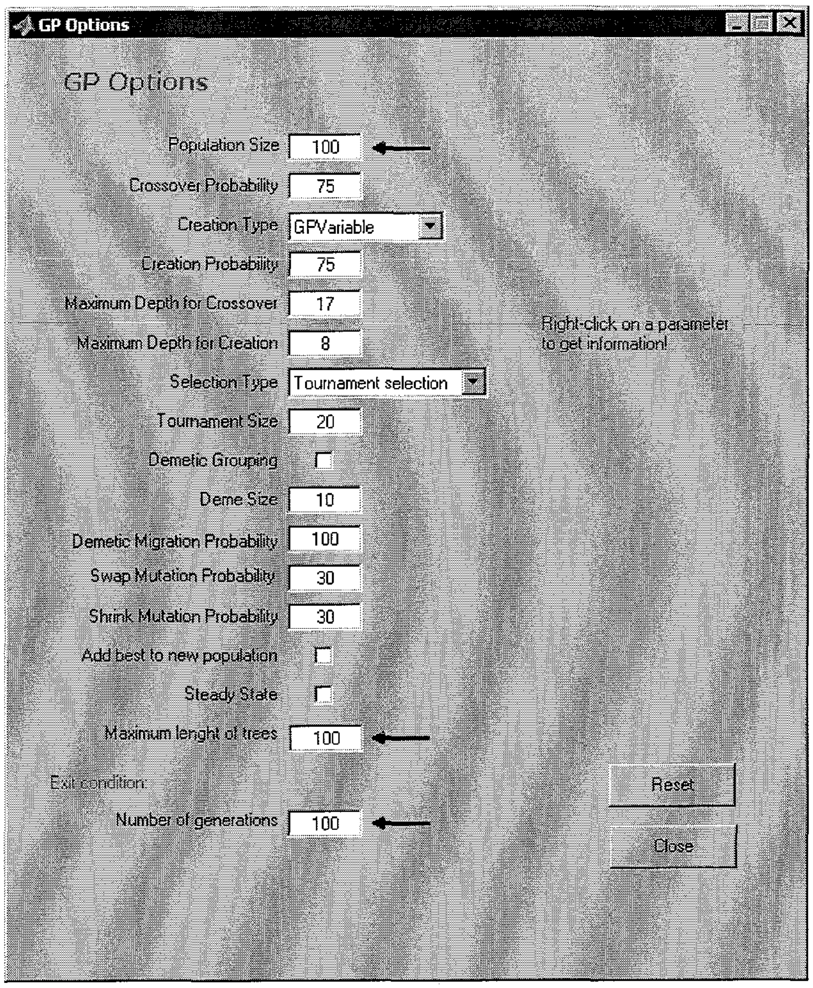
In this window the options for the GP run can be set. If you are not familiar with GP, your best option is to push the 'reset' button (in the lower-right corner of the window). This will set the parameters to default values that will be sufficient to solve most problems. Three parameters are of interest: the population size, the number of generations and the maximum length of the trees (all three are indicated with arrows in Figure 5.2). These three parameters are directly related to the speed at which the LyFF will execute, but are also dependent on the complexity of the problem that is being submitted. For simple problems these values can be kept low, for more complex problems it is advised to raise the values. Generally, the best thing to do is to try different values, and get a feel for how these parameters influence your calculation. For more information about all parameters and how they were implemented in the GP Kernel, see [8].

5.2.3 Lyapunov Options Window

This window is used to set the fitness cases and some options concerning the Lyapunov function search. The algorithms are already described in Chapter 4. As we will see, parts of the algorithm can be turned on and off, as desired. Again, we will go through the options step by step, the numbers corresponding to the numbers in Figure 5.3.

1. First we start defining the fitness cases on the Coarse Grid. These can be set by defining a value for the Radius and for the Number of Points. By the checking the Rectangular Form-box, the form of the Coarse Grid will change from a circle to a rectangular form. The Coarse Grid cannot be turned on or off.

2. To create a Fine Grid check the Grid-box. Again, enter values for the Length and the Number of Points. Also, there is an option to create an even more dense grid around the origin, the Additional Grid. Some problems can benefit from using this extra grid.



GP Options

Population Size	100
Crossover Probability	75
Creation Type	GPVariable
Creation Probability	75
Maximum Depth for Crossover	17
Maximum Depth for Creation	8
Selection Type	Tournament selection
Tournament Size	20
Demetic Grouping	☐
Deme Size	10
Demetic Migration Probability	100
Swap Mutation Probability	30
Shrink Mutation Probability	30
Add best to new population	☐
Steady State	☐
Maximum lenght of trees	100

Exit condition:

Number of generations	100

Right-click on a parameter to get information!

Figure 5.2: GP Options Window

3. Every time a parameter from the previous options is changed, the resulting grid is visualized here.

4. To maximize the Region of Attraction, as described in Section 4.2.3, check this box.

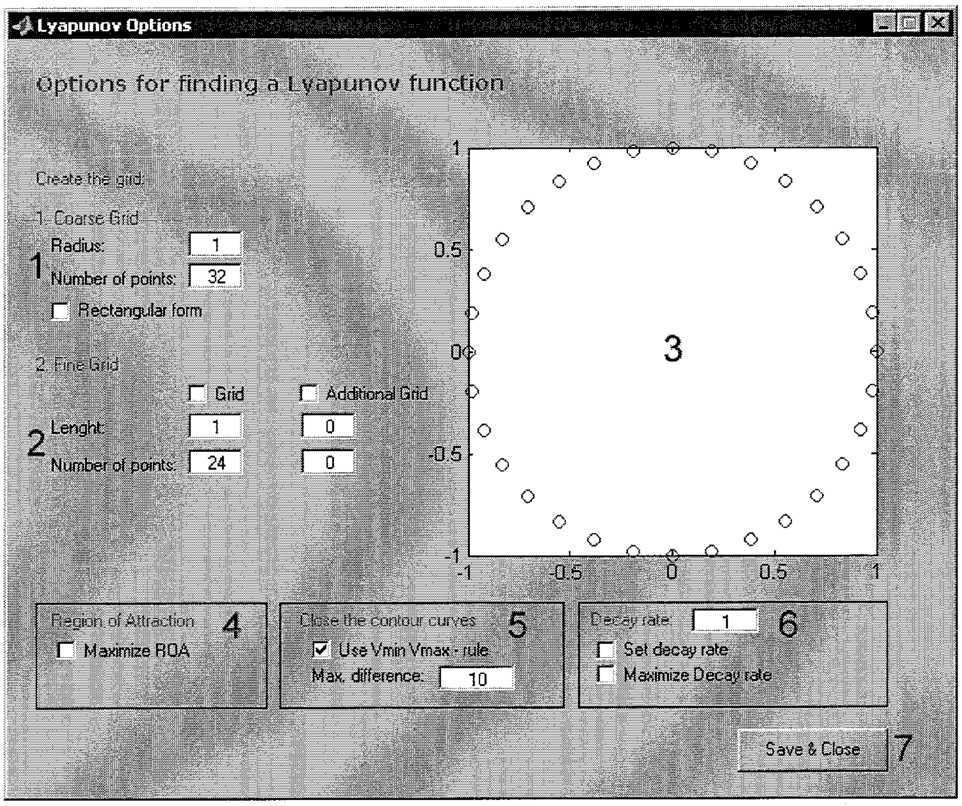
5. To try to close the contour curves as described in Section 4.2.2, check this box. Also, the maximum difference allowed between V_{max} and V_{min} should be entered here.

6. For the Decay Rate there are two options:

    - Set the Decay Rate to a minimum value.
    - Try to optimize the Decay Rate.

    In both cases a value has to be entered. In the first case, this is a minimal value for the decay rate: during a run the individuals that have a Decay Rate below this value will be punished, otherwise they won't be punished. This way we hope to find an individual that has at least the set value for the Decay Rate. In the second case, the value is the value from where we want to start the optimization of the decay rate. When an individual is found that has an equal or higher Decay Rate than the set Decay Rate, the value for the Decay Rate is raised and the search starts over again. The value of the new Decay Rate is written to the input file during the run.

7. To close this window and return to the main window, press the Save & Close-button


Lyapunov Options

Options for finding a Lyapunov function

Create the grid:

1. Coarse Grid

Radius: 1

Number of points: 32

☐ Rectangular form

2. Fine Grid

☐ Grid ☐ Additional Grid

Lenght: 1 0

Number of points: 24 0

Region of Attraction

☐ Maximize ROA

Close the contour curves

☑ Use Vmin Vmax - rule

Max. difference: 10

Decay rate: 1

☐ Set decay rate

☐ Maximize Decay rate

Figure 5.3: Lyapunov Options Window

5.3 Post-processing using Matlab

After the run has ended, Matlab can use the configuration file to examine the results. Different functions have been written to simplify this. In Matlab help is available for these functions by typing help <function>.

5.3.1 Functions for Post-processing

The following functions can be used to examine the result of LyFF.

Checking the grid

Function syntax: checkgrid(F, np, r)

F	a function (sym-object)
np	number of points
r	radius

Checkgrid can be used to check a function on a (very fine) grid. It will create a grid and for all grid-points evaluate the function. As a result a graph will be shown, indicating values that are > 0 with green dots, values < 0 with red dots, and values $= 0$ with black dots.

Checking the decay rate

Function syntax: Decay(Vx, dVx, Grid) or Decay(Vx, dVx, np, r)

Vx	the Lyapunov function (sym-object)
dVx	the derivative of the Lyapunov function (sym-object)
Grid	matrix that represents a grid: the first column is x_1, the second x_2 etc.
np	number of points
r	radius

Decay is used to calculate the minimal value of the decay rate on a given grid. You can either enter the grid in matrix form, or enter the number of points and radius. Decay will calculate the values of the decay rate on every point of the grid and will return the minimal value.

Calculating the region of attraction

Funtion syntax: roa(F, Grid) or roa(F, np, r)

F	a function (sym-object)
Grid	matrix that represents a grid: the first column is x_1, the second x_2 etc.
np	number of points
r	radius

Roa calculates the largest region of attraction within a given grid. You can enter the grid either in matrix form, or enter the number of points and radius. Roa will not create a real grid, but instead it creates points on the edge of the grid (so if you manually enter a grid, be sure that the points are on the edge of your search domain). The points on the edge of the grid are needed to calculate the minimal value of $V(\mathbf{x})$ on the edge of the grid. Then the largest region of attraction is calculated and also the largest box that fits within this region of attraction as defined in Section 4.2.3. Then a plot of the region of attraction and the box will be drawn.

Making a contour plot

Function syntax: vxcontour(F, r, t, plot)

F	a function (sym-object)
r	radius
t	value for which the contourline of $V(\mathbf{x})$ should be plotted. If t is not given all contourlines will be drawn. Note that t can also be a vector, to draw multiple contourlines
plot	for plotting numbers with the contourlines, else 0. Default: plot = 1

Vxcontour will draw a contourplot of function F.

Creating a vector plot

Function syntax: createvector(dx1, dx2, r)

dx1, dx2	system
r	radius

Createvector will create a vector plot of the system $f(dx1, dx2)$. An easy way to use this function is to load fun.mat in the Matlab workspace, which will load dx1 and dx2. Then, run this function.

5.3.2 Additional Functions

The following functions cannot be directly used to examine the results, but are used in the GUI and by the functions described in the previous section.

Creating a grid

Function syntax: [x,y] = CreateGrid(np,r)

x,y	coordinates of grid points
np	number of points
r	radius

Creategrid creates a grid that is needed to find a Lyapunov function, i.e. it creates a grid that omits the origin.

Creating the rectangular grid

Function syntax: [x,y] = CreateRecGrid(np, r)

x,y	coordinates of grid points
np	number of points
r	radius

Createrecgrid creates a rectangular grid of points that lie on the edge of the given radius, ensuring that points on the axes are taken in as well.

Calculating $\dot{V}(\mathbf{x})$

Function syntax: lyap2dv(dx1, dx2, Vx)

dx1, dx2	system
Vx	the Lyapunov function

As the system and the Lyapunov function are known, using the function Lyap2dv is a convenient way to calculate the derivative of the Lyapunov function.

The vector plot

Function syntax: quivernew(...)

This function is in essence the same as the original `quiver`-command in Matlab, however it differs in that it will plot all vectors with the same length. This function is called by createvector.

Resetting GP options

Function syntax: resetfun

No inputs.

This function will reset the GP options in fun.mat to its default values.

Saving to file

Function syntax: savetofile(varname, varvalue)

varname	the name of the variable you wish to save
varvalue	the value of the variable you wish to save

In the GUI many variables are to be saved, this function simplifies that. By calling this function with its arguments, it will save the variables name and value to fun.mat. Multiple variables can be saved at once by placing them in succession, e.g.: savetofile(x1, 10, x2, 20).

5.4 Implementation of the Algorithm in the GP Kernel

The original GP Kernel can be found on the internet in the public domain. In [8] the features of the GP Kernel are described. To implement the Lyapunov problem, some of the original files were modified and files were added. These files are outlined here, to facilitate later changes of the program. The code of these files can be found in the Appendix.
A list of the modified and added files:

```
fparser.cpp         grids.h            printfile.cpp       LyFF.h
fparser.h           lyapvars.cpp       printfile.h
grids.cpp           lyapvars.h         LyFF.cpp
```

5.4.1 The Main File

The most important part of the program is the file where the main-function resides, in this case in `LyFF.cpp`. A listing of the code with explanatory comments can be found in Appendix A. Some of the concepts are explained here:

Reading the MAT-file

At the start of the program the configuration file (either created manually or by the GUI) is opened, various parameters are read, and put in memory for later use. This is done as follows: a structure is globally created in which for each variable an entry is declared that contains the following data:

- the name (a string) under which it was saved in the configuration file.
- the type of data (datastring, dataint, datadouble or datagrid). The first three types are the regular C++-types string, integer and double. The grid type is a type that contains values to create a grid. See Section 5.4.3 for more information on grids.
- a pointer to the variable in the program. This variable should already have been declared in the program!

For example:

```
char *dx1, *dx2;

struct GPConfigVarInformation configArray[]=
{
    {"dx1c", DATASTRING, &dx1},
    {"dx2c", DATASTRING, &dx2},
    {"", DATAINT, NULL}
}
```

will try to find the variable with the name dx1c in the configuration file. If it is found it will get the value of this variable and save it as a string in the variable dx1. The last line in the structure, with the pointer *NULL*, will tell the program to stop searching for parameters. If a parameter is not found, a warning will be given at the start of the program. Most variables have a default value, so if a variable is not present in the MAT-file the program will still be able to run. However, variables that are not defined can have a fatal effect on the execution of the program!

If you want to add parameters, you only need to define a new variable and add a line of code to the structure.

The actual reading of the file is programmed in the module config.cpp. This module was modified from the original to open the MAT-file. The code listing can be found in Appendix E.

Saving values to the configuration file

Function syntax:
`void SaveToMFile(char *MFile, char *Var, double Vard, char *Name, int Type)`

To examine results, we need to write these to the configuration file. As several parameters are written back to the file, a function has been designed to do this. Two types of data can be written: a double or a string. The function is called with the following parameters:

- MFile. The MAT-file where we want to write the data to.
- Var and Vard. The data that needs to be saved. In the case of a string, we save the string in Var, and we assign a zero to Vard. In the case of a double, we assign an empty value, i.e. " ", to Var and the desired value of the double to Vard. This is a bit complicated and, for convenience, should be changed in the future.
- Name. The name we wish to give to the variable in the MFile.
- Type. The data type: DATASTRING or DATADOUBLE.

Evaluating an individual

Function syntax:
`double MyGene::evaluate(grids x, int i, MyGP& gp, double arg0, double arg1)`

Because the individuals are saved in a tree-form, we need a function to evaluate them. This function takes care of that. It is a recursive function, meaning that it will continue to call itself until the tree has been completely traversed. Every function- and terminal-evaluation are programmed here. So when a function or terminal is added, be sure to add its evaluation here too!

Calculating $V(\mathbf{x})$ and $\dot{V}(\mathbf{x})$

Function syntax:
`void CalcVx(MyGene *pa, MyGP &pb, grids Grid, std::valarray<float> &Array)`
`void CalcdVx(MyGene *pa, MyGP &pb, grids Grid, std::valarray<float> &Array)`

pa	A pointer to the starting node of the tree.
pb	The address of self, which is the individual.
Grid	The grid for which we want to calculate $V(\mathbf{x})$.
Array	The address of the array in which the values of $V(\mathbf{x})$ should be saved.

As we are going to calculate the values of $V(\mathbf{x})$ for all the points on a grid, and also for multiple grids, a function has been designed to do this quickly. The calculation of $V(\mathbf{x})$ is not very difficult, as we only need to evaluate the individual at the given Grid. The calculation of $\dot{V}(\mathbf{x})$ is somewhat more difficult, but has already been explained in Section 4.2.1.

Calculating the fitness

Function syntax:
`void MyGP:: evaluate()`

This is the essential part of the whole program. It calculates the fitness of an individual using the algorithm as described in Chapter 4.

Defining functions and terminals

Function syntax:
void createNodeSet(GPAdfNodeSet &adfNs)

This function takes care of the creation of the node set, i.e. the set of functions and terminals. It is pretty straightforward, so it is fairly easy to add a node. However, don't forget to declare space for an extra node!

5.4.2 The Lyapunov Class

The Lyapunov Class is created to hold all Lyapunov related parameters. It provides a back-up: if the user does not provide a value for a parameter, a default value is entered. It also provides an easy way for printing the variables to a stream. The code can be found in Appendix B.

5.4.3 The Grids Class

Because we use multiple grids, and also some specific functions to operate on these grids, a special class has been created.

An object of the class *grids* contains two arrays, x_1, and x_2, that are publicly accessible. These arrays are filled with the coordinates of the fitness cases.

There are two ways to define an object of the class *grids*. The first way is to define it without passing parameters. Only the object is defined then, without initializing the arrays. The arrays can later be initialized using the function declare(). Second, an object can be defined by passing a mxArray. The mxArray should consist of a matrix, that should be built up of the vectors $[x_1, x_2]$, representing the coordinates of the fitness cases.

Although parts of the Grids class are capable of dealing with 3-dimensional grids, LyFF in its entirety is only able to solve 2-dimensional problems. However, expanding LyFF to enable it to be used for multi-dimensional problems is not very hard. Some of the functions in the class will have to be redefined, and the calculation of $V(\mathbf{x})$ and $\dot{V}(\mathbf{x})$ will have to be adjusted.

Within the class the following functions have been defined:

- a copy constructor = is defined, for making copies of an object.
- The function declare(const mxArray M) can be used to change the values of the arrays x_1 and x_2, after the object is already declared. The mxArray M should satisfy the same requirements as given above.
- The function disturb(grids A, int state, float dx) is used during the calculation of $\dot{V}(\mathbf{x})$. A new grid is created, copied from the original grid A, and subsequently the value dx is added to the chosen array state.
- Because arrays are used, they should be deleted after use. This is done by the function makeMT().
- The function create(int np, int ns) is for internal use of the class. It creates a new object with *ns* number of dimensions and reserves space for the arrays of length np.

The listing of the interface and implementation can be found in Appendix C.

5.4.4 The Print File

As the individuals are saved in a tree-form during the run, we need some specialized printing functions to be able to print the individuals to different files. Also, as we will see later, we will want to print to text-files as well as LaTeX-files. The implementation of these printing functions can be found in Appendix D. Remember that when you add a function or a terminal, you should also add it to these printing functions!

5.4.5 The Parser

To dynamically enter formulas, a parser needs to be used to parse the formulas from a text string to a workable formula for the program. The parser used is open source code, found on the internet [9]. The only thing that needs to be kept in mind is that the parser only accepts variables of text length 1, e.g. x and y. So, as the GUI accepts larger strings of variables, e.g. $x1$ or $x2$, the formulas entered in the GUI need to be converted. This is done in the code of the GUI. If the GUI is not used, one should enter the formulas in a text string in the MAT-file, using x and y as variables.
The manual of the parser can be found in Appendix F.

5.5 The Configuration File Format and Output Files

For LyFF to work, it needs a configuration file with the settings for the run. During the run, some parameters are written back to this configuration file. Furthermore, three files are created with statistics of the run. All files will be explained in this section.

5.5.1 The Configuration File and its Format

The configuration file is used to pass data from Matlab to LyFF and vice versa. The format used is the standard MAT-format. In Table 5.1 all variables are listed, with their respective data types and an indication whether or not they are used by LyFF. Note that the data type indication is referring to data types as defined in C++, with the exception of the type grid, that is defined in the grid class, and the type switch, which will be explained in a moment. Matlab does not make a distinction between different numeric types, so integers, doubles, grids, switches and booleans are all saved as double arrays in the MAT-file. Strings are saved as char arrays which can be converted to symbolic objects. However, it is important to note the C++ types as the parameters will be converted to the indicated types as they are read by LyFF.
Note that:

- Not all parameters are used for configuration of LyFF. Some are just here in the file, so that the next time you open the GUI, its last state is recreated.
- Some parameters are yes/no parameters, i.e. whether an option should or should not be used during a run. The values are booleans: yes = 1, and no = 0.
- Some parameters offer multiple options. These are indicated with a switch data type and correspond to the following options:
  - `Creation type`

0	Probabilistic selection
1	Tournament selection

  - `Selection type`

0	GPVariable
1	GPGrow
2	GPRampedHalf
3	GPRampedHalfVariable
4	GPRampedGrow

These options are further explained in [8].

- The additional grid is not saved as a separate grid, but integrated in the Fine Grid.
- The parameter NumberOfStates is not used at the moment, as LyFF solely works for 2-dimensional problems.
- The parameter dVx is created by the post-processing function lyap2dv after the run.

The following parameters are created by LyFF during the run:

- The parameter Elapsed Time gives the time it took for the run to end in seconds.
- The parameter Gen indicates in which generation the run was stopped.
- When optimizing the decay rate, the parameter GenDecay indicates in which generation the value for the decay rate was last updated.
- When maximizing the region of attraction, the parameter GenROA indicates in which generation the value for the region of attraction was last updated.
- The parameter Vx is a string with the best-of-run individual. The post-processing causes this parameter to be converted to a Matlab sym-object. The original string is saved to Vxorg.

Table 5.1: Listing of variables in MAT-file

Name	Data type	LyFF
`AddBestToNewPopulation`	boolean	x
`AddGrid`	boolean	
`AddGridNP`	integer	
`AddGridRadius`	double	
`CircleGrid`	grid	x
`CoarseGrid`	grid	x
`CreationProbability`	boolean	x
`CreationType`	switch	x
`CrossoverProbability`	double	x
`DecayRate`	boolean	x
`DemeSize`	boolean	x
`DemeticGrouping`	boolean	x
`DemeticMigProbability`	double	x
`ElapsedTime`	double	x
`FineGrid`	grid	x
`Gen`	integer	x
`GenDecay`	integer	x
`GenROA`	integer	x
`InfoFileDir`	string	x
`InfoFileName`	string	x
`InfoTitle`	string	x
`MaxDecay`	boolean	x
`MaxLength`	integer	x
`MaxROA`	boolean	x
`MaxRelDif`	double	x
`MaximumDepthForCreation`	integer	x
`MaximumDepthForCrossover`	integer	x
`NumberOfGenerations`	integer	x

Table 5.1: continued.

Name	Data type	LyFF
NumberOfPointsCoarse	integer	
NumberOfPointsFine	integer	
NumberOfPointsGrid	integer	
NumberOfStates	integer	
PopulationSize	integer	x
RadiusCoarse	double	
RadiusFine	double	
RadiusGrid	double	
RecForm	boolean	
SelectionType	switch	x
ShrinkMutationProbability	double	x
SteadyState	boolean	x
SwapMutationProbability	double	x
TestGrid	boolean	
TournamentSize	integer	x
VMinMax	boolean	x
ValDecayRate	double	x
Vx	string	x
Vxorg	string	
dVx	string	
dx1	string	
dx1c	string	x
dx2	string	
dx2c	string	x
use_add	boolean	x
use_con	boolean	x
use_cos	boolean	x
use_div	boolean	x
use_mul	boolean	x
use_pow	boolean	x
use_sin	boolean	x
use_sub	boolean	x

5.5.2 Output Files

Before startup of a run, a copy is made of the configuration file fun.mat and is named <InfoFileName>.mat and placed in the directory as entered in <InfoDirName>. This provides an easy way of back-up of the configuration per run.

During the run three files are created, namely: <InfoFileName>.stc, <InfoFileName>.dat and <InfoFileName>.tex. They are placed in the directory as entered in <InfoDirName>. The .stc-file gives statistical information about the run, i.e. the best, average and worst fitness, length and depth of the individuals per generation. The .dat-file saves the best-of-generation individual per generation, as does the .tex-file but the latter is in LaTeXformat.

5.6 Known Errors and ToDo List

Although the program is capable of running complex problems, it is still not finished completely. In this paragraph a list of known errors and points that could be improved is given.

5.6.1 Known Errors

In a few cases, the program is known to behave in an unpredictable way. These cases are described here.

- The program has not been thoroughly tested on different operating systems. All tests described in this report were performed using Windows 95, where they ran smoothly. A few runs showed no obvious problems using Windows 98 either. However, a quick check on one machine running Windows NT resulted in a crash of the program after approximately 15 runs. Although no further testing was done on other NT-machines, we suspect that it was caused by a memory leak in the program. This still has to be fixed.
- The functions roa and lyap2dv in Matlab are computational very demanding. For very complex problems these functions are known to halt during execution. More computer memory should solve this problem.

5.6.2 ToDo List

The following features could be improved, as that would make the program more user friendly and more stable.

- LyFF is only capable of handling 2-dimensional problems. However, due to the implementation of the grids-class it is very easy to extend this to multi-dimensional problems.
- A numerical approximation of $\dot{V}(\mathbf{x})$ is used during a run. By implementing a analytical differentiator, the error induced by the approximation can be eliminated.
- There is no type checking of the entered information in the GUI. If a user provides data of an incorrect type, e.g. puts a string in a numeric field, the outcome of the run is undefined.
- A GUI for the post-processing needs to be created.
- Opening the GUI loads fun.mat, regardless of the current directory. So if, at startup of the GUI, the user is in another directory than that where fun.mat is located, the GUI will open with blank fields. As the user starts entering data, a new fun.mat will be created in the current directory. However, if the run is started, the GUI will not be able to find LyFF.exe, and the DOS-box will not open. To avoid this problem, the user must navigate to the directory containing the file LyFF.exe, before starting the GUI. This problem still has to be corrected.

Chapter 6

Results

To test the GP algorithm, several problems, ranging from simple to more difficult, have been submitted to the algorithm. The results are compared to other methods for finding Lyapunov functions, and are presented here. Calculations are performed on a 500 MHz Pentium III processor with 128 Mb internal memory, running Windows 95.

6.1 GP Variables

For all runs, unless otherwise mentioned, the following GP variables are used:

```
CrossoverProbability          = 75
CreationProbability           = 75
CreationType                  = GPRampedHalf
MaximumDepthForCreation       = 8
MaximumDepthForCrossover      = 17
SelectionType                 = Tournament Selection
TournamentSize                = 20
DemeticGrouping               = Off
DemeSize                      = 10
DemeticMigProbability         = 100
SwapMutationProbability       = 30
ShrinkMutationProbability     = 30
AddBestToNewPopulation        = No
SteadyState                   = No

GP tree 0: +(2) -(2) *(2) ^2(1) x1 x2
```

Some of the variables are options that can be switched on and off for a run: 'no' meaning that that option was not used, 'yes' that it is used during the run. Unless otherwise mentioned, all Lyapunov function are tested on the domain $X^0 = [-1, 1] \times [-1, 1] \subset \mathcal{R}^2$.

All results are displayed in figures and tables. In the tables only $V(\mathbf{x})$ is given. Furthermore, the region of attraction r and sometimes the decay rate α are also given in the table. The decay rate α is calculated by creating a grid of 100.000 points on X^0, and then calculating $\min(-\frac{\dot{V}(\mathbf{X})}{V(\mathbf{x})})$. The largest region of attraction is determined by drawing the largest closed curve within X^0 and then fitting in a box as explained in Section 4.2.3.
The last value given is *gen*, which is the generation in which the particular Lyapunov function candidate was found.

6.2 A Very Simple Problem

Given the system

$$\begin{aligned} \dot{x}_1 &= -x_1 \\ \dot{x}_2 &= -x_2 \end{aligned} \tag{6.1}$$

a trivial Lyapunov function is

$$\begin{aligned} V(\mathrm{x}) &= x_1^2 + x_2^2 \\ \dot{V}(\mathrm{x}) &= -x_1^2 - x_2^2 \end{aligned} \tag{6.2}$$

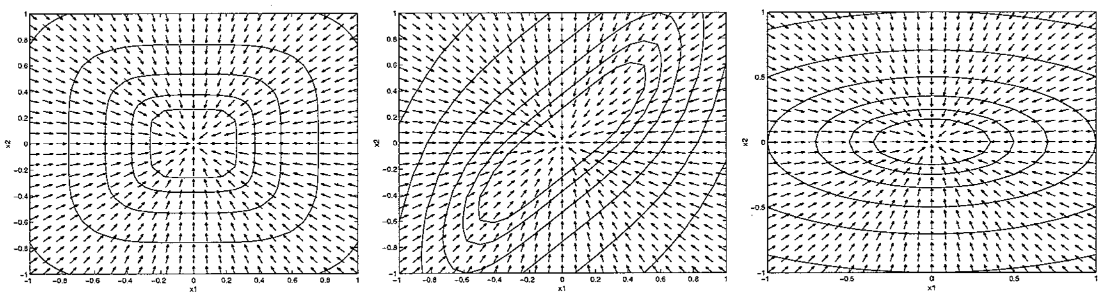
Test 1

Using the following settings, GP came up with the results as shown in Table 6.1 and Figure 6.1. Often, when searching for a Lyapunov function, humans tend to start trying quadratic functions. GP, however, is not biased and just searches for a function that meets the conditions. And as there are many more possibilities than only the quadratic functions, the runs resulted in more 'exotic' functions. All functions are Lyapunov functions.

```
PopulationSize                  = 100
NumberOfGenerations             = 100, or stop if fitness = 0
Number of points on Circle grid = 48
Number of points on Fine grid   = 100
VMin-Max Rule                   = No
MaxROA                          = No
Decay Rate                      = No
Files                           = vsp1a, vsp1b, vsp1c
```

Table 6.1: Results of Test 1

$V(\mathrm{x})$	r	α	gen
$V_1 = x_1^4 + x_2^4$	0.84	4	1
$V_2 = ((x_2 - x_1)^2 + x_1^4)^2$	0.27	4	1
$V_3 = 4x_2^2 + x_1^2$	0.45	2	1



(a) V_1 (b) V_2 (c) V_3

Figure 6.1: Results for Test 1

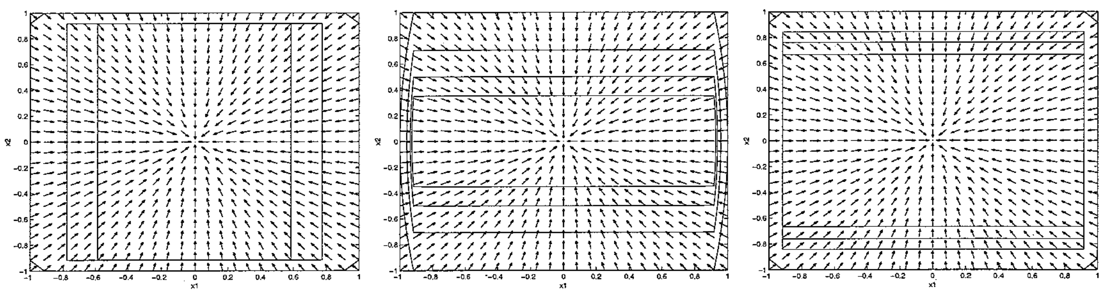
Test 2 (Maximizing Region of Attraction)

Now we are going to try to maximize the region of attraction. The results are shown in Table 6.2. The largest region of attraction in X^0 corresponds to a contour curve with the form of a square that follows the edge of the domain. As can be seen in Figure 6.2, this is exactly the form that GP tried to approximate.

`PopulationSize`	`= 100`
`NumberOfGenerations`	`= 100`
`Number of points on Circle grid`	`= 48`
`Number of points on Fine grid`	`= 100`
`VMin-Max Rule`	`= No`
`MaxROA`	`= Yes`
`Decay Rate`	`= No`
`Files`	`= vspROA1a, vspROA1b, vspROA1c`

Table 6.2: Results of Test 2

$V(\mathbf{x})$	r	α	gen
$V_1 = x_2^{2048} + x_1^{32}$	0.96	32	39
$V_2 = x_1^{68} + x_2^{2}$	0.93	2	36
$V_3 = x_2^{256} + x_2^{32} + x_1^{1024} + x_1^{16384}$	0.96	32	81



(a) V_1 (b) V_2 (c) V_3

Figure 6.2: Results for Test 2

6.3 The Variable Gradient Method

This method constructs a Lyapunov function by assuming a certain form for the gradient of an unknown Lyapunov function, and then finding the Lyapunov function itself by integrating the assumed gradient. For low order systems, this approach sometimes leads to the successful discovery of a Lyapunov function [6].

Given the nonlinear system

$$\begin{aligned} \dot{x}_1 &= -2x_1 \\ \dot{x}_2 &= -2x_2 + 2x_1x_2^2 \end{aligned} \tag{6.3}$$

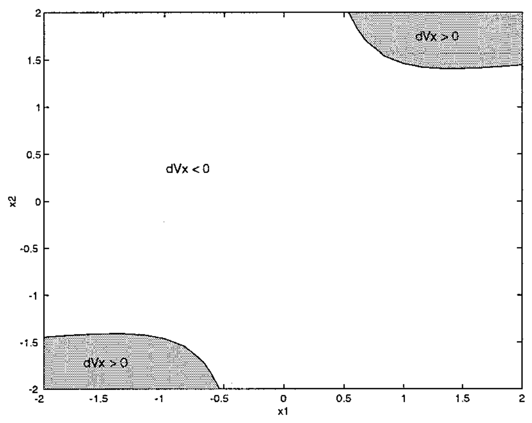
the following Lyapunov function is found using the Variable Gradient Method [see 6,p. 87]:

$$\begin{aligned} V(\mathbf{x}) &= \tfrac{x_1^2+x_2^2}{2} \\ \dot{V}(\mathbf{x}) &= -2x_1^2 - 2x_2^2(1 - x_1x_2) \end{aligned} \tag{6.4}$$

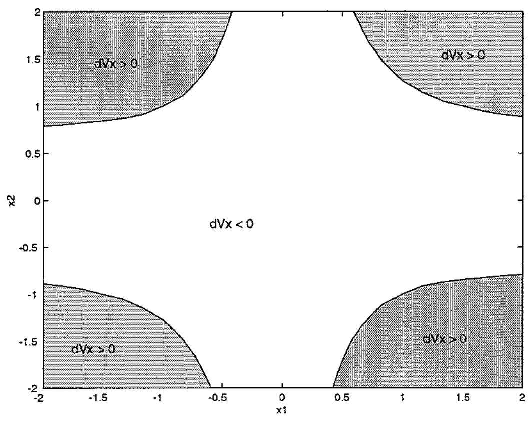
Another Lyapunov function, also found with the Variable Gradient Method:

$$V(\mathbf{x}) = \frac{x_1^2}{2} + \frac{3}{2}x_2^2 + x_1 x_2^3 \tag{6.5}$$

In Figure 6.3 the regions where $\dot{V}(\mathbf{x}) > 0$ are visualized for both functions. As $V(\mathbf{x})$ is positive definite and $\dot{V}(\mathbf{x})$ is locally negative definite, asymptotic stability is locally guaranteed.



(a) Plot of regions of $\dot{V}(\mathbf{x})$, Eq. (6.4)



(b) Plot of regions of $\dot{V}(\mathbf{x})$, Eq. (6.5)

Figure 6.3: Solutions using the Variable Gradient Method

Test 3

Using GP with the following settings:

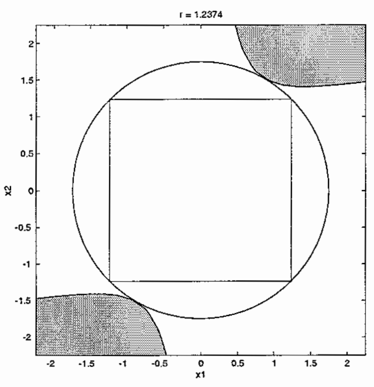
```
PopulationSize                 = 100
NumberOfGenerations            = 100, or stop if fitness = 0
Number of points on Circle grid = 28
Number of points on Fine grid  = 64
VMin-Max Rule                  = No
Decay Rate                     = No
Files                          = vgm1-1 / vgm1-9
```

The results are shown in Table 6.3 and Figure 6.4.

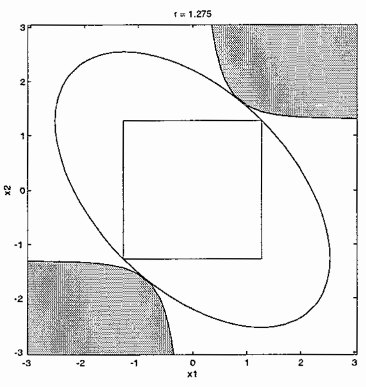
Table 6.3: Results of Test 3

$V(\mathbf{x})$	r	gen
$V_1, V_{10} = x_1^2 + x_2^2$	1.24	1
$V_2 = x_1^2 + x_1 x_2 + x_2^2$	1.28	1
$V_3 = x_2^4 + x_1^2 + x_2^8$	1.04	1
$V_4 = (x_1^6 + 2x_1^5 x_2 + x_1^4 x_2^2 + x_1^2 + 2x_2 x_1 + x_2^2 + 17x_2^4$ $-32x_2^5 + 24x_2^6 - 8x_2^7 + x_2^8 + 2x_2^7 x_1^2 - 4x_2^6 x_1^3 + 2x_2^5 x_1^4 - 2x_2^3 x_1 + x_2^2 x_1^2)^2$	0.68	1
$V_5 = 2x_1^2 + 2x_1 x_2 + x_2^8$	1.26	2
$V_6 = 2x_1^2 - 4x_1 x_2 + 2x_1 x_2^2 + 4x_2^2 - 4x_2^3 + x_2^4$	0.21	1
$V_7 = (x_2^2 - 2x_2 x_1 + x_1^2 + x_1^4)^2$	0.40	1
$V_8 = 2x_1^2 + x_2^2$	1.21	1
$V_9 = x_2^4 + x_2^2 + 2x_1 x_2 + x_1^2$	1.16	1

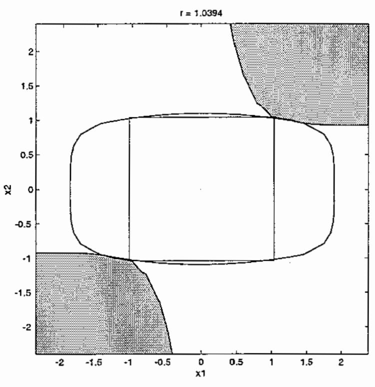
Almost all results are obtained within on average 1 run. This is very fast, and it indicates that during startup of the run more than enough diverse individuals were created, because the result emerged already after initialization.



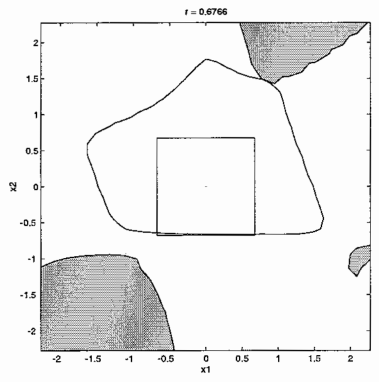
(a) V_1, V_{10}



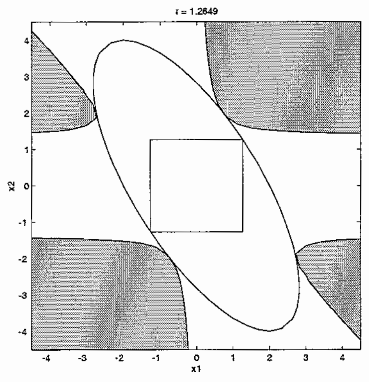
(b) V_2



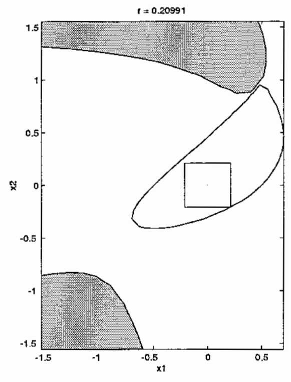
(c) V_3



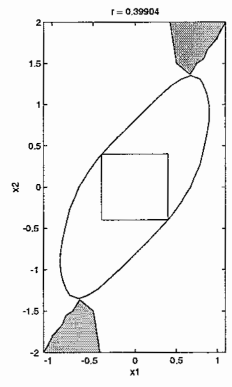
(d) V_4



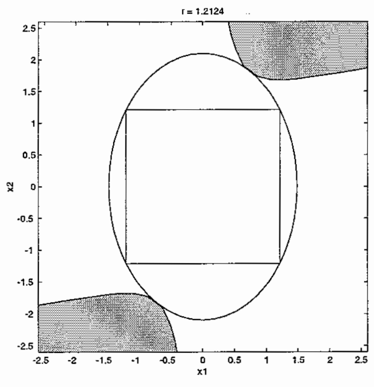
(e) V_5



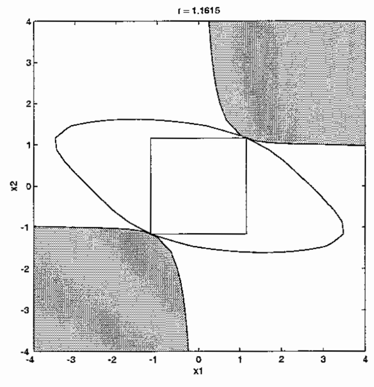
(f) V_6



(g) V_7



(h) V_8



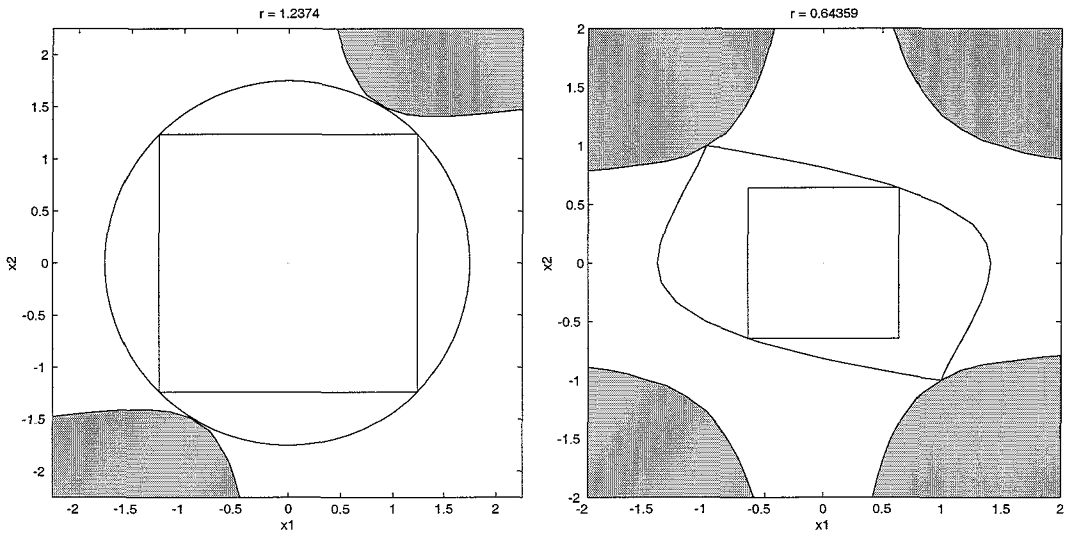
(i) V_9

Figure 6.4: Results of Test 3. The grey areas indicate the areas where $\dot{V}(\mathbf{x}) > 0$.

Evaluation afterwards showed that all functions are Lyapunov functions that prove local stability. Shown in Figure 6.4 are the largest regions of attractions. These are determined visually, by first plotting the $\dot{V}(\mathbf{x}) = 0$ lines and then fitting in the largest closed contour curve of $V(\mathbf{x})$. Then, a box is drawn within the region of attraction (as explained in Section 4.2.3), to calculate its size. Note that although the run was set to search within the domain $X^0 = [-1, 1] \times [-1, 1]$, some solutions are able to prove stability for a larger region.

Test 4 (Maximizing Region of Attraction)

The region of attraction of the Lyapunov functions found using the Variable Gradient method are shown in Figure 6.5.



(a) Region of attraction of Eq. (6.4) (b) Region of attraction of Eq. (6.5)

Figure 6.5: Region of attraction. The grey areas indicate the areas where $\dot{V}(\mathbf{x}) > 0$.

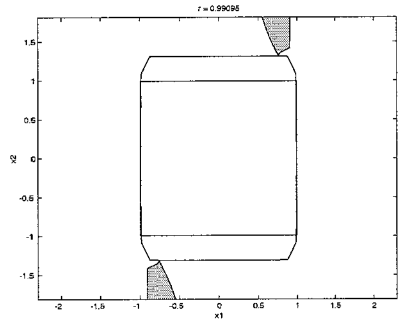
Five tests were done with the following settings:

```
PopulationSize                  = 100
NumberOfGenerations             = 100
Number of points on Circle grid = 48
Number of points on Fine grid   = 100
Decay Rate                      = No
MaxROA                          = Yes
Files                           = vgmroa1-1/vgmroa1-4
```

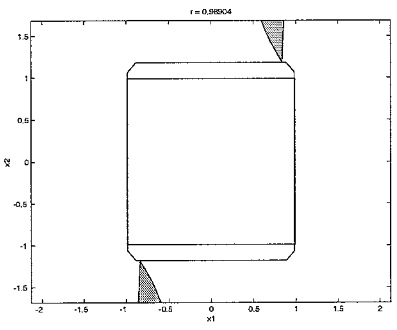
The results are shown in Table 6.4. All functions are Lyapunov functions and prove that the origin is a asymptotically stable equilibrium point. However, the region of attraction is not larger than found in the previous tests. This is probably due to the fact that the search domain is $X^0 = [-1, 1] \times [-1, 1]$. Eq. 6.4 indicates that the system is locally stable, the region is indicated in Figure 6.3. The domain that we have tested for lies within this region. Therefore we enlarge the search domain to $X = [-1.5, 1.5] \times [-1.5, 1.5] \subset \mathcal{R}^2$.

Table 6.4: Results of test 4

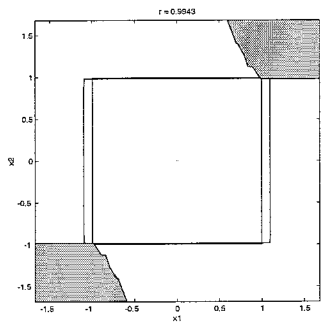
Function	r	gen
$V_1 = x_1^{34} + x_2^2$	0.99	22
$V_2 = x_2^2 + x_1^{80}$	0.99	10
$V_3 = x_2^{272} + x_1^2$	0.99	10
$V_4 = ((x_2 - x_1)(x_2 + x_1)x_2^2 + x_1^4 + (x_1^4 + x_2^4)^2)^2$	1.35	25
$V_5 = x_1^2 + x_2^{144}$	0.95	15



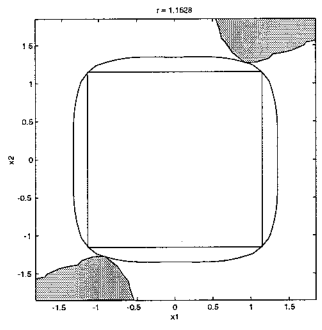
(a) V_1



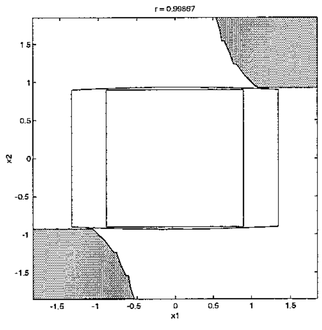
(b) V_2



(c) V_3



(d) V_4



(e) V_5

Figure 6.6: Region of attraction of Test 4

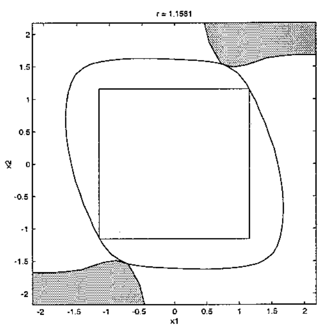
Test 5 (Maximizing the Region of Attraction 2)

The search domain is enlarged to $X^0 = [-1.5, 1.5] \times [-1.5, 1.5]$.

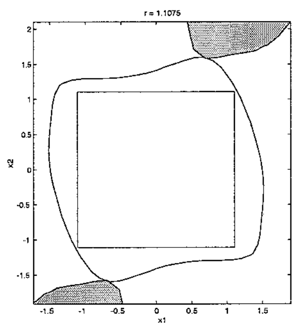
```
PopulationSize                  = 100
NumberOfGenerations             = 100
Number of points on Circle grid = 48
Number of points on Fine grid   = 100
Decay Rate                      = No
MaxROA                          = Yes
Files                           = vgmroa3-1/vgmroa3-5
```

Table 6.5: Results of Test 5

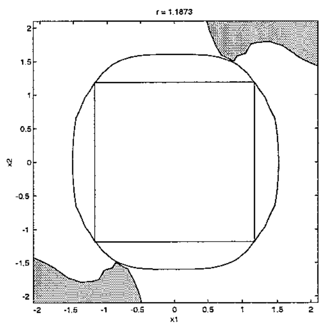
Function	r	gen
$V_1 = 4x_1^6 + 3x_1^2x_2^2 + x_2^4(x_2+x_1)^2 + 2x_1^2 + 2x_2^8 + (x_1^2 + 2x_2(x_2+x_1))^2x_1^4$	1.16	95
$V_2 = ((x_2+x_1)^2 + (x_2-x_1)(x_2+x_1) - x_1x_2 + 4x_1^2)^2 + (2(x_2-x_1)(x_2+x_1) - x_1x_2+$ $((x_2-x_1)(x_2+x_1) - x_1x_2)^2)^2$	1.11	72
$V_3 = ((x_1^4x_2^2 + x_2^2)^2 + 2x_1^4 + 3x_2^2 - x_1^2)^2 + (x_1^4 + 6x_2^2 + 2x_2^4 + (x_1^4 + x_2^2)^2 + x_1^4x_2^2)^2$	1.19	100
$V_4 = (4x_2^2 + (2x_1^4 + x_2^2)x_2^2 + 2x_1^4 + (2x_1^4 - x_1^2 + x_2^2)(x_1^4 - x_1^2 + x_2^2))^2$	1.16	79
$V_5 = (x_1^4x_2^2 + 2x_2^2 - x_1^2)^2 + x_1^8 + x_2^2$	1.22	40



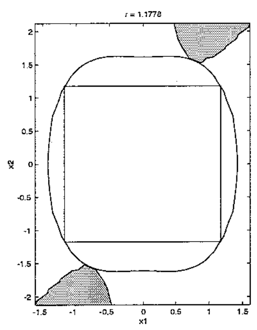
(a) V_1



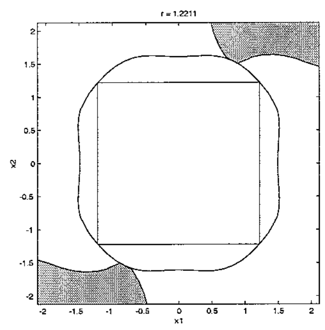
(b) V_2



(c) V_3



(d) V_4



(e) V_5

Figure 6.7: Results of Test 5

Now, the results are somewhat better than in the previous test, but the runs took more generations. However, the largest region of attraction is not larger than the one found with the Variable Gradient Method. This is due to the method of maximizing the region of attraction: before the region of attraction is maximized a function must be found that fulfills $V(\mathbf{x}) > 0$ and $\dot{V}(\mathbf{x}) < 0$ on the whole domain X^0. Therefore, the region of attraction that is found depends very much on the choice of X^0. If we would enlarge the search domain to $X^0 = [-2, 2] \times [-2, 2]$ a larger region of attraction would be found, although the search for it would be harder, as fewer functions fulfill $V(\mathbf{x}) > 0$ and $\dot{V}(\mathbf{x}) < 0$ on this domain. The new method proposed in Section 4.2.3 should result in better solutions, as this method automatically enlarges the search domain.

6.4 Krasovskii's Method

Krasovskii's method suggests a simple form of Lyapunov function candidates for autonomous nonlinear systems of the form (3.1), namely $V = \mathbf{f}^T\mathbf{f}$. The basic idea of the method is simply to check whether this particular choice indeed leads to a Lyapunov function [6].

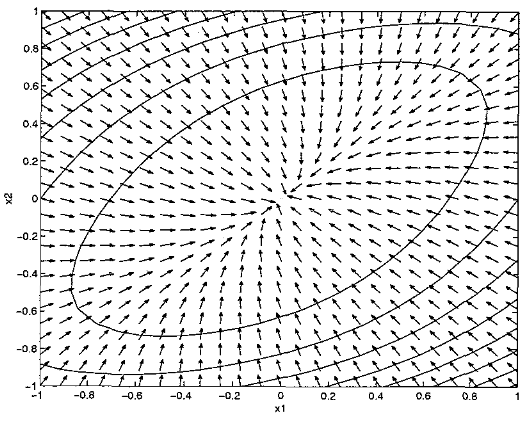
For the following system:

$$\begin{aligned}\dot{x}_1 &= -6x_1 + 2x_2 \\ \dot{x}_2 &= 2x_1 - 6x_2 - 2x_2^3\end{aligned} \tag{6.6}$$

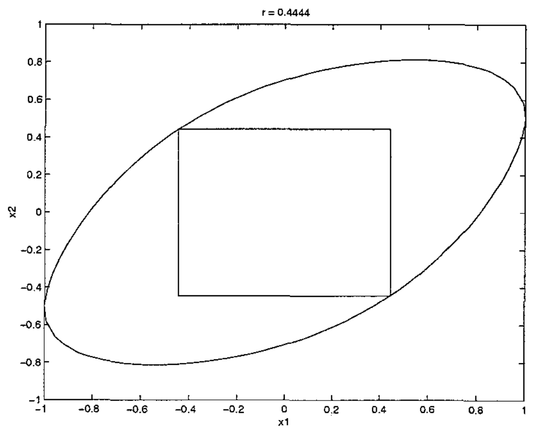
the proposed Lyapunov function, found with Krasovskii's method, is:

$$V(\mathbf{x}) = (-6x_1 + 2x_2)^2 + (2x_1 - 6x_2 - 2x_2^3)^2 \tag{6.7}$$

In Figure 6.8 the contour plot and the region of attraction of Eq. 6.7 are drawn. The decay rate $\alpha = 8$.



(a) Contour and vector plot of Eq. 6.7



(b) Region of Attraction of Eq. 6.7

Figure 6.8: Properties of Eq. 6.7

Test 6

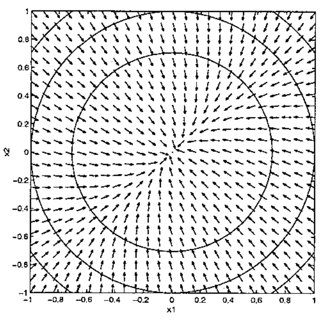
Using the following settings, LyFF is started to find Lyapunov function candidates:

```
PopulationSize                   = 100
NumberOfGenerations              = 100, or stop if fitness = 0
Number of States                 = 2
Number of points on Circle grid  = 48
Radius Circle                    = 1
Number of points on Fine grid    = 64
Radius Grid                      = 1
Test Grid                        = Yes
VMin-Max Rule                    = No
Decay Rate                       = No
Files                            = kras1-1/kras1-10
```

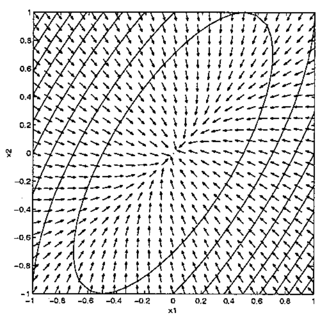
Results are shown in Table 6.6 and Figure 6.9. Within the domain $X^0 = [-1,1] \times [-1,1]$ the Lyapunov function candidates are all Lyapunov functions. When testing on the domain $X^0 = [-2,2] \times [-2,2]$ all candidates are still Lyapunov functions, except for V_6 and V_{10}, see Figure 6.10.

Table 6.6: Results for Test 6

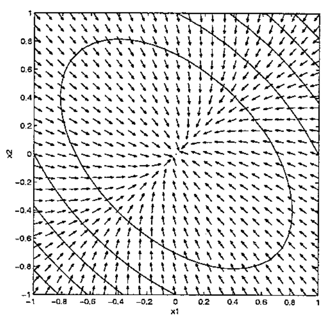
Function	r	α	g
$V_1, V_4, V_7 = x_1^2 + x_2^2$	0.71	8.00	1
$V_2 = 2x_1^2 - 2x_1x_2 + 2x_2^2$	0.32	7.53	1
$V_3 = 2x_1^2 + 2x_1x_2 + 2x_2^2$	0.50	8.00	1
$V_5 = x_2^4 + x_1^2 + 2x_1x_2 + x_2^2$	0.27	7.99	1
$V_6 = (-x_2^2 + 2x_1x_2 - 6x_1^2 + x_1^2x_2^2)^2$	0.29	8.28	3
$V_8 = 4x_2^2 + x1^2$	0.45	7.00	1
$V_9 = x_1^2 + x_2^2 + x_2^8$	0.69	8.00	1
$V_{10} = (x_1^3x_2 + x_2^2 + x_1^2)^2$	0.57	3.57	1



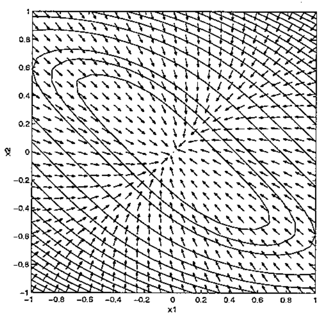
(a) V_1, V_4, V_7



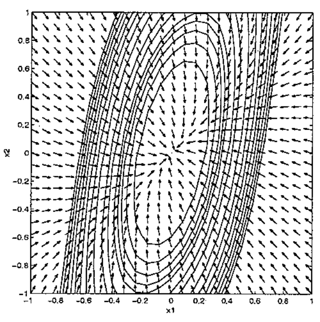
(b) V_2



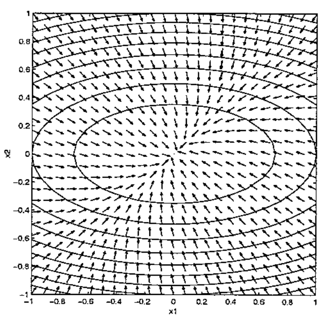
(c) V_3



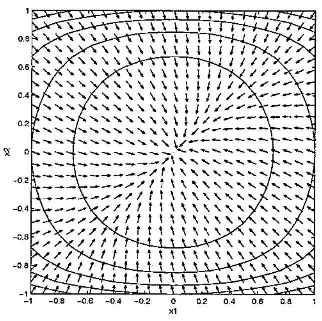
(d) V_5



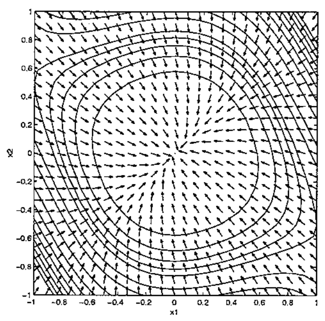
(e) V_6



(f) V_8

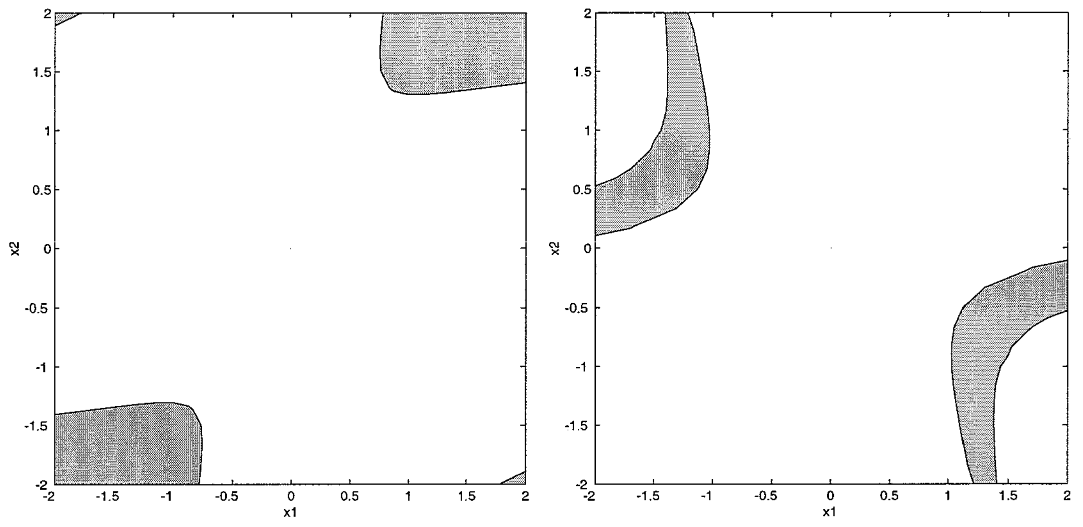


(g) V_9



(h) V_{10}

Figure 6.9: Results of Test 6



(a) $\dot{V}_6$ (b) $\dot{V}_{10}$

Figure 6.10: Results of Test 6 for $\dot{V}_6$ and $\dot{V}_{10}$. The grey areas indicate the areas where $\dot{V}(\mathbf{x}) > 0$

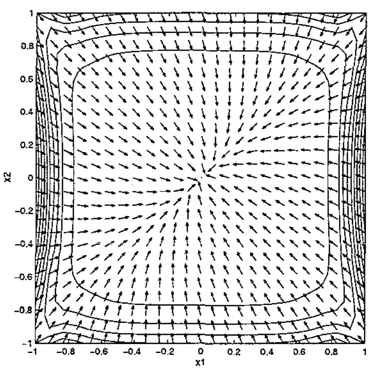
Test 7 (Maximizing the Region of Attraction)

```
Population Size                  = 100
Number Of Generations            = 100
Maximum Length of Trees          = 300
Number of points on Circle grid  = 48
Number of points on Fine grid    = 400
VMin-Max Rule                    = No
Decay Rate                       = No
Maximize Region of Attraction    = Yes
Files                            = krasroa2-1/krasroa2-3
```

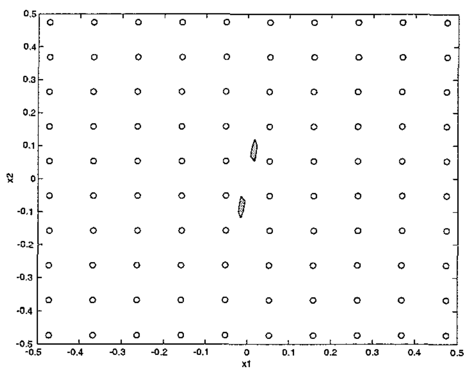
The results of three runs are shown in Table 6.7 and Figure 6.11. Both V_1 and V_3 are Lyapunov functions in X^0, V_2 is not. The phenomenon as described in Section 4.3.1 occurs: $\dot{V}(\mathbf{x}) > 0$ between the grid points, as shown in Figure 6.4. The region of attraction of the Lyapunov function generated by Krasovskii's method was 0.44. The region of attraction of the Lyapunov functions generated by LyFF are clearly larger.

Table 6.7: Results of Test 7

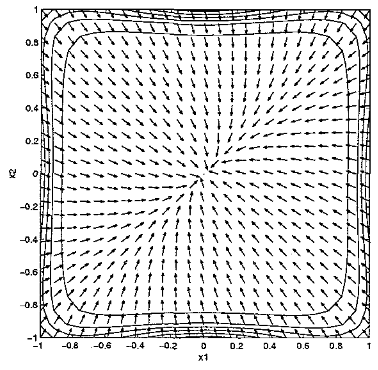
Function	r	α	gen
$V_1 = x_1^4 + x_2^4 + (x_1^4 - x_2^2)^2(x_1^4 + (x_1^2 - x_1^8 + x_1^4)^4 + x_2^2)^2$	0.85	16	30
$V_2 = x_1^2 + x_2^{64} + x_2^4 + (2x_1^2 + (x_1^2 + x_2^{32})^2 + x_1^{64} + 2x_2^{32} + x_1x_2^4)(x_1^2 + x_2^{64})(x_1^2 + x_2^{32})$	-	-	20
$V_3 = (4x_2^2 + (x_2^2 + (x_2^4 - x_1^2)^2)^2 + 2x_1^4 + ((x_1^2 - x_2^2)^2 + x_2^2 + x_1^2)^4 + x_2^2(x_2^4 - x_1^2)^2 +$ $x_1^2 + x_2^4)(2(x_1^2 - x_2^2)^2 + 2x_2^2 + 2x_1^2 + (x_2^4 - x_1^2)^2)$	0.94	15.53	7



(a) V_1



(b) V_2



(c) V_3

Figure 6.11: Results of Test 7

6.5 Johansens Problem

In [7] Johansen describes how Lyapunov functions can be found using convex optimization. The basic idea is to assume a linear parameterization of the set of Lyapunov function candidates. The existence of a Lyapunov function leads to two linear inequalities that must hold for every $x \in X$. By discretizing the compact set X, the possible Lyapunov functions within the selected class of candidates are characterized (approximately due to discretization) by a finite number of linear inequalities. Hence, the problem is reduced to a convex optimization problem involving linear inequality constraints at each point in the state space. By discretization of the state space this leads to a computational procedure based on linear or quadratic programming that is effective for systems of sufficiently low order [7].

The procedure is illustrated using a numerical example. For the problem

$$\begin{aligned} \dot{x}_1 &= -3x_1 + x_2 \\ \dot{x}_2 &= \frac{2x_1^2}{0.3+(x_2+0.4)(x_2-0.6)} - 2x_2 \end{aligned} \tag{6.8}$$

several parameterizations are proposed and Lyapunov functions candidates are computed while optimizing the decay rate or the region of attraction. Johansen does not mention the actual functions that he found, but gives information about the values he obtained for the decay rate or the region of attraction (see Table 6.8). Therefore comparison between GP generated Lyapunov function and Lyapunov function found using convex optimization is done using these values.

Table 6.8: Best Results using Convex Optimization

Objective	r	α
Decay Rate	0.30	5.06
Region of Attraction	0.85	1

Test 8 (No optimization)

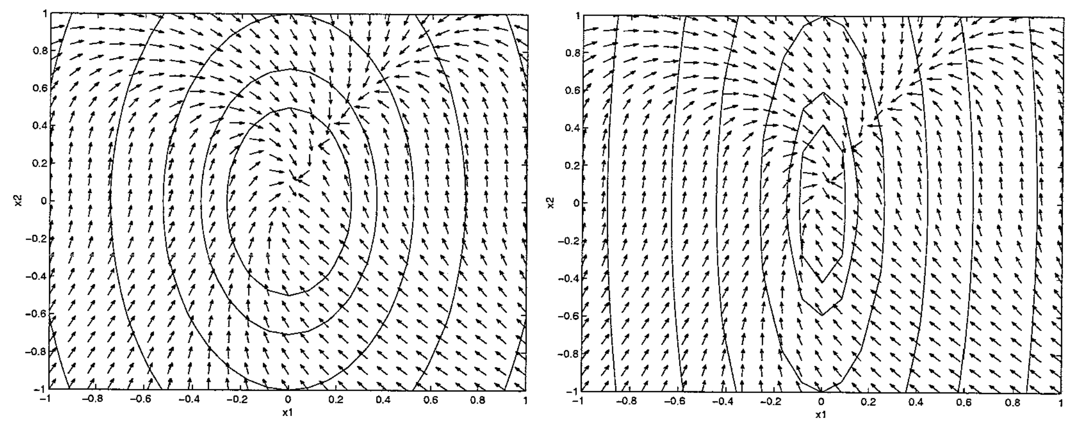
For this test, we are simply trying to find a Lyapunov function on X^0. Three candidates are generated using the following settings:

```
Number of Individuals              = 500
Maximum length of trees            = 300
Number of points on Circle grid    = 32
Number of points on Fine grid      = 256
DecayRate                          = No
Maximize Region of Attraction      = No
Files                              = joh2, joh2a, joh2b
```

The results are shown in Table 6.9 and Fig. 6.12. Solution V_2 is no valid Lyapunov candidate, as its decay rate α is negative, meaning that $\dot{V}(\mathbf{x}) > 0$ somewhere in X.

Table 6.9: Results of test 8

$V(\mathbf{x})$	r	α	gen
$V_1 = 58x_1^2 + 16x_2^2$	0.46	0.99	5
$V_2 = (764x_1^2 + 45x_2^2 - 2x_1)^2$	0.23	-99	21
$V_3 = (x_2^2 + 14x_1^2)^2$	0.26	1.46	5



(a) V_1 (b) V_3

Figure 6.12: Results of Test 8

Test 9 (Maximizing the region of attraction)

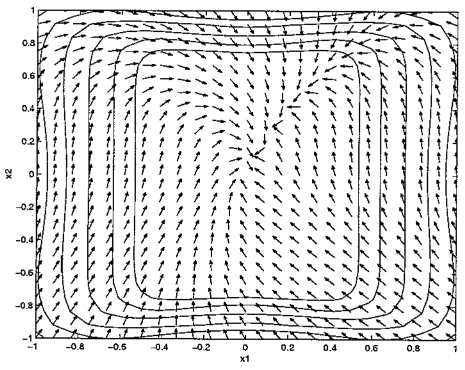
Now, we set the algorithm to maximizing the region of attraction. Furthermore, we demand that during the run that, at worst, $\alpha = 1$. If $\alpha < 1$ a penalty is given, but otherwise nothing will happen (the decay rate will not be optimized). All runs run for 100 generations, then they are cut off, the best of run being the one with the largest region of attraction.

```
Number of Individuals              = 500
Maximum length of trees            = 300
Number of points on Circle grid.   = 32
Number of points on Fine grid      = 256
DecayRate                          = No
Maximize Region of Attraction      = Yes
Files                              = joh-roa1a, joh-roa1c, joh-roa3, joh-roa1d
```

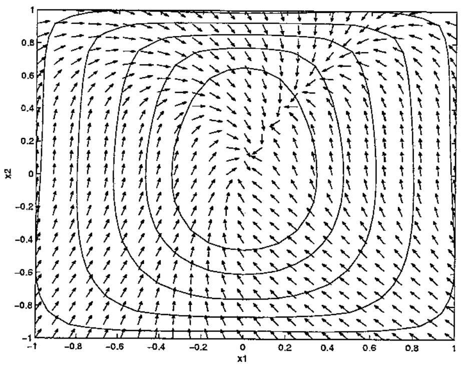
Table 6.10: Results of Test 9

$V(\mathbf{x})$	r	α	gen
$V_1 = ((x_2^2 - x_1^2)^2 + x_2^2 - x_1^2)^2 + ((x_1^4 + (x_2^2 - x_1^2)^2 + x_2^4)^2 + x_2^4 + x_2^8)^2 + x_1^8 + 16x_1^4$	0.92	2.53	64
$V_2 = 5x_1^4 + 12x_1^2 + (x_1^2 + x_2^2)^2 + x_2^2 + (x_2^2 - x_1^8 - 2x_2)^2 + (x_2^8 + (x_2^4 - x_1^2 + x_2^2)^2 + x_1^2)^2 + (x_1^2 + x_1^4)^2$	0.89	0.77	27
$V_3 = (4x_2^8 + x_2^2)^4 + (x_1^4 + 4x_1^2)^4$	0.94	5.01	15
$V_4 = (((x_2^2 - x_2^2x_1^2 + x_1^4)(x_2^2 - x_1^2) + x_2^2)^2 + x_2^2)^2 + 16x_1^4$	0.89	2.46	39

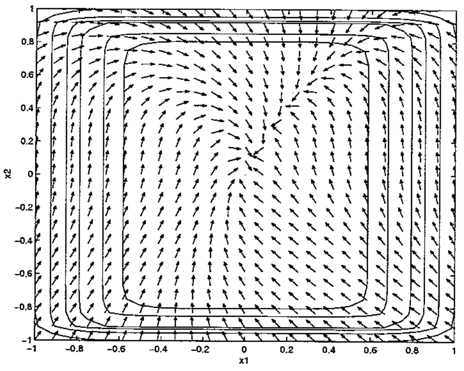
In Table 6.10 the results are given. All candidates are Lyapunov functions. Compared to the results published in [7] all solutions have a larger region of attraction, but V_2 has a smaller decay rate. V_3 in particular is a very good result, as its decay rate is near the maximal value that was obtained using the convex optimization method.



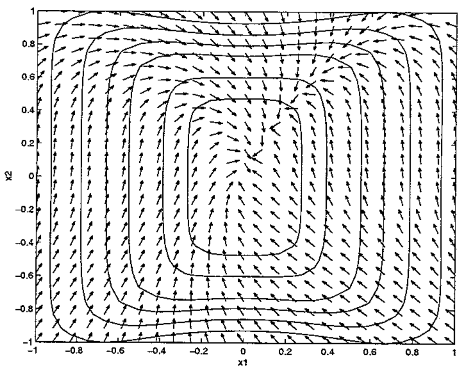
(a) V_1



(b) V_2



(c) V_3



(d) V_4

Figure 6.13: Results for Test 9

Test 10 (Maximizing the Decay Rate)

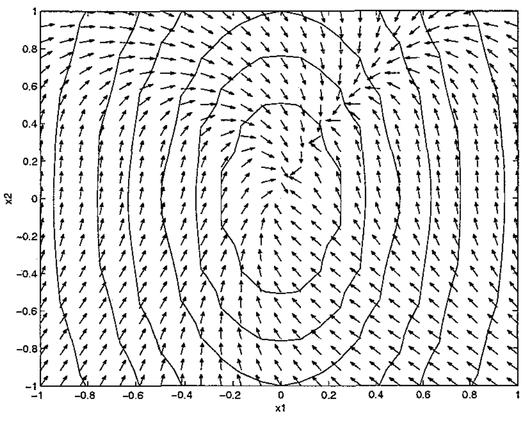
The objective of this test is to find a Lyapunov function that maximizes the decay rate.

```
Number of Individuals            = 500
Maximum length of trees          = 300
Number of points on Circle grid  = 48
Number of points on Fine grid    = 768
DecayRate                        = Yes
Maximize Region of Attraction    = No
Files                            = joh-decay1, joh-decay2b, joh-decay2c
```

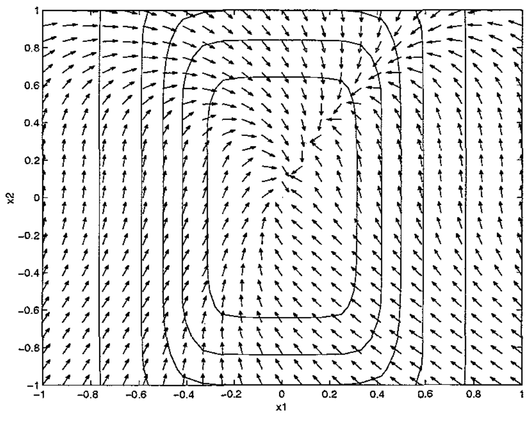
The results are shown in Table 6.11 and Figure 6.14. Again, all candidates are Lyapunov functions and the results are better than the results obtained using convex optimization.

Table 6.11: Results of Test 10

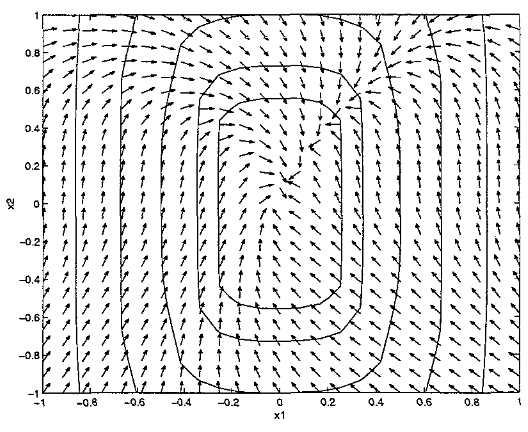
$V(\mathbf{x})$	r	α	gen
$V_1 = (4x_1^2 + x_2^2)^8$	0.43	9.18	unknown
$V_2 = 256x_1^8 + (x_1^4 + x_2^4)^2$	0.50	5.00	18
$V_3 = (x_2^4 + 16x_1^4)^4$	0.49	10.20	18



(a) V_1



(b) V_2



(c) V_3

Figure 6.14: Computed Lyapunov functions for Test 10

Chapter 7

Conclusions And Future Research

Finding a Lyapunov function and thereby proving (local) stability of the equilibria of a non linear system is not an easy task, but GP can lighten the task. Admittedly, not every run ends up in a Lyapunov function, but one does not have to go into complex mathematical calculations to obtain a Lyapunov function, nor does one need to know much about GP since a GUI in Matlab is now available for easy implementation for the search for a Lyapunov function. Only validation of the Lyapunov function is needed afterwards.
Unfortunately, due to the complexity of some of the GP generated solutions analytical validation is not always possible, and one has to revert to numerical methods. Although a method for calculating a sufficiently dense grid size has been proposed, it has not yet been used to validate the Lyapunov candidates.

The GP algorithm has been compared to several other methods for finding Lyapunov functions. In these tests the GP algorithm repeatedly proved to be able to obtain Lyapunov functions that were better in the sense of the region of attraction or the decay rate. On the other hand, the results were often very complex functions that were difficult to validate. A way to prevent this, is to severely limit the depth to which the algorithm may create functions. But, in limiting the depth of functions, we might also be limiting the ability to find functions that have such large regions of attraction or decay rates.

Future research could include the use of this GP algorithm during identification of systems. In previous research we tried to dynamically identify a physical system using GP. This resulted in a mathematically correct function, which in simulation turned out to be an unstable system. By applying LyFF during the identification process, and thereby ascertaining stability of the systems, we suspect that this problem can be overcome. Furthermore, LyFF could also be used to investigate the stability of discontinuous systems. For these class of systems Lyapunov functions are very hard to find analytically.

All in all, we can conclude that GP is useful in the search for Lyapunov functions. In a broader perspective we can conclude that GP is becoming a very interesting tool for solving complex problems. We showed that by translating a problem in a fitness function a hard nonlinear problem could be solved. Finding controllers or identifying complex systems are similar difficult problems, and we expect that these problems can also be solved using GP.

Bibliography

[1] A.M. Lyapunov. *The General Problem of Motion Stability.* 1892.

[2] D.G. Schultz and J.E. Gibson. *The Variable Gradient Method for Finding Lyapunov Functions*, volume 81 of *AIEE Trans. part II, Appl. & Industry*, pages 203–210. 1962.

[3] N.N. Krasovskii. *Problems of the theory of stability of motion.* Stanford Univ. Press, Stanford, 1963. translation of the Russian edition, Moskow (1959).

[4] I.A.C. Soute, M.J.G. van de Molengraft, and G.Z. Angelis. Using genetic programming to find lyapunov functions. 2001.

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

[6] J.E. Slotine and W. Li. *Applied Nonlinear Control.* Prentice Hall, 1991.

[7] Tor A. Johansen. Computation of lyapunov functions for smooth nonlinear systems using convex optimization. 1999.

[8] A. Fraser and T. Weinbrenner. *The Genetic Programming Kernel Version 0.5.2*, 1997.

[9] Warp. *Function Parser for C++ by Warp.* http://www.students.tut.fi/ warp/FunctionParser/.

Appendix A

The Main File

The Implementation: LyFF.cpp

```
/* -------------------------------------------------------------

Lyapunov function finder
------------------------------------------------------------ */
#include <conio.h> // _kbhit
#include <iostream>
#include <fstream.h>
#include <strstrea.h>
#include <iomanip.h>
#include <time.h>

#include <stdlib.h>
#include <new.h>    // For the new-handler
//#include <math.h>
#include <string.h>
#include "fparser.h"

// Include header file of genetic programming system.
#include "gp.h"
#include "gpconfig.h"

// Include headerfiles of my functions
#include "symbreg.h"
#include "grids.h"
#include "lyapfunction.h"
#include "lyapvars.h"

#include "mex.h"
#include "mat.h"

#include <valarray>

// Globally declared, because we need them in several functions.
int gen;
grids CoarseGrid, FineGrid;  // For more info on the grids-object, see grids.h and grids.cpp
```

```
// This is the mat file from and to which all variables are read/written.
char *MFile="fun.mat";
FunctionParser Func1;
FunctionParser Func2;

// We use this parameter to keep a tab wether a solution has been
// written to the output file.
int OutFlag = 0;
float minFit;

// Functions to be used during GP run. 0 = not used, 1 = using.
// If you should want to add, more functions, start by adding them here.
int useADD = 0,
    useSUB = 0,
    useMUL = 0,
    useDIV = 0,
    useCON = 0,
    useSIN = 0,
    useCOS = 0,
    usePOW = 0;

// Define configuration parameters for the GP run in the GPVariables. If you need more
// variables, just add them below and insert an entry in the configArray.
// Lyapunov variables can be added to lyapvars.h and lyapvars.cpp
GPVariables cfg;
LyapVariables cfgl;

char *dx1, *dx2;


// Parameteres are loaded as follows:
// {"nameMAT", TYPE, var}
// File is opened, and the variable nameMAT is looked up in the file
// The value is then saved to var of type TYPE.
// See config.cpp for more info.

struct GPConfigVarInformation configArray[]=
{
    {"dx1c", DATASTRING, &dx1},
    {"dx2c", DATASTRING, &dx2},
    {"MaxDecay", DATAINT, &cfgl.MaxDecay},
    {"MaxROA", DATAINT, &cfgl.MaxROA},
    {"DecayRate", DATAINT, &cfgl.DecayRate},
    {"ValDecayRate", DATADOUBLE, &cfgl.ValDecayRate},
    {"MaxRelDif", DATADOUBLE, &cfgl.MaxRelDif},
    {"CoarseGrid", DATAGRID, &CoarseGrid},
    {"FineGrid", DATAGRID, &FineGrid},
    {"MaxLength", DATAINT, &cfgl.MaxLength},
    {"InfoFileName", DATASTRING, &cfgl.InfoFileName},
    {"InfoFileDir", DATASTRING, &cfgl.InfoFileDir},
    {"InfoTitle", DATASTRING, &cfgl.InfoTitle},
    {"PopulationSize", DATAINT, &cfg.PopulationSize},
    {"NumberOfGenerations", DATAINT, &cfg.NumberOfGenerations},
    {"VMinMax", DATAINT, &cfg1.VMinMax},
    {"CreationType", DATAINT, &cfg.CreationType},
    {"CrossoverProbability", DATADOUBLE, &cfg.CrossoverProbability},
    {"CreationProbability", DATADOUBLE, &cfg.CreationProbability},
    {"MaximumDepthForCreation", DATAINT, &cfg.MaximumDepthForCreation},
    {"MaximumDepthForCrossover", DATAINT, &cfg.MaximumDepthForCrossover},
    {"SelectionType", DATAINT, &cfg.SelectionType},
    {"TournamentSize", DATAINT, &cfg.TournamentSize},
    {"DemeticGrouping", DATAINT, &cfg.DemeticGrouping},
    {"DemeSize", DATAINT, &cfg.DemeSize},
    {"DemeticMigProbability", DATADOUBLE, &cfg.DemeticMigProbability},
    {"SwapMutationProbability", DATADOUBLE, &cfg.SwapMutationProbability},
    {"ShrinkMutationProbability", DATADOUBLE, &cfg.ShrinkMutationProbability},
    {"SteadyState", DATAINT, &cfg.SteadyState},
    {"AddBestToNewPopulation", DATAINT, &cfg.AddBestToNewPopulation},
    {"use_mul", DATAINT, &useMUL},
    {"use_sub", DATAINT, &useSUB},
    {"use_add", DATAINT, &useADD},
    {"use_div", DATAINT, &useDIV},
    {"use_con", DATAINT, &useCON},
    {"use_sin", DATAINT, &useSIN},
    {"use_cos", DATAINT, &useCOS},
    {"use_pow", DATAINT, &usePOW},
    {"", DATAINT, NULL}
};


// This function is the divide with closure. Basically if you divide
// anything by zero you get an error so we have to stop this
// process. We check for a very small denominator and return a very
// high value.
inline double divide (double y1, double y2)
{
  if (fabs (y2)<1e-6)
    {
      if (y1*y2<0.0)
    return -1e6;
      else
    return 1e6;
    }
  else
    return y1/y2;
}


// Easy way to save variables to the MATfile
void SaveToMFile(char *MFile, char *Var, double Vard, char *Name, int Type)
{
    MATFile *pmat;
    mxArray *pa1;
    int status;

    //open matfile
    pmat = matOpen(MFile, "u");
    if(pmat == NULL) cout << "Error opening functions-file\n";

    switch(Type)
    {
    case DATASTRING:        //string
        pa1 = mxCreateString(Var);
        mxSetName(pa1, Name);
        break;
    case DATAINT:
    case DATADOUBLE:        //scalar
        pa1 = mxCreateScalarDouble(Vard);
        mxSetName(pa1, Name);
        break;
    }

    //Put mxArray in file
    status = matPutArray(pmat, pa1);
    if (status !=0) cout << "Error writing array to file./n" ;

    //destroy matlab arrays!
    mxDestroyArray(pa1);

    //close MatFile
    status = matClose(pmat);
    if (status != 0) cout << "Error closing file\n";
}


// We have the freedom to define this function in any way we like. In
// this case, it takes the parameter x that represents the terminal X,
// and returns the value of the expression. It's recursive of course.

// If you add functions, don't forget to implement them here!!
// Also, don't forget to implement them in printfile.cpp!!
double MyGene::evaluate (grids x, int i, MyGP& gp, double arg0, double arg1)
{
    double ret, a0, a1;

    if (isFunction ())
        switch (node->value ())
        {
            case '*':
                ret=NthMyChild(0)->evaluate (x, i, gp, arg0, arg1)
                * NthMyChild(1)->evaluate (x, i, gp, arg0, arg1);
                break;
            case '+':
                ret=NthMyChild(0)->evaluate (x, i ,gp, arg0, arg1)
                  + NthMyChild(1)->evaluate (x, i ,gp, arg0, arg1);
                break;
            case '-':
                ret=NthMyChild(0)->evaluate (x, i, gp, arg0, arg1)
                  - NthMyChild(1)->evaluate (x, i, gp, arg0, arg1);
                break;
            case '^':
                ret=pow(NthMyChild(0)->evaluate (x, i, gp, arg0, arg1),2);
                break;
            case 's':
                ret=sin(NthMyChild(0)->evaluate (x, i, gp, arg0, arg1));
                break;
            case 'c':
                ret=cos(NthMyChild(0)->evaluate (x, i, gp, arg0, arg1));
                break;
            case '%':
  // We use the function divide rather than "/" to ensure the
  // closure property
                ret=divide (NthMyChild(0)->evaluate (x, i, gp, arg0, arg1),
                    NthMyChild(1)->evaluate (x, i ,gp, arg0, arg1));
                break;
            case 'A':
  // This is the ADF0 function call.  We have access to that
  // subtree, as the GP gave us a reference to itself as
  // parameter.  We first evaluate the subtrees, and then call
  // the adf with the parameters
                a0=NthMyChild(0)->evaluate (x, i, gp, arg0, arg1);
                a1=NthMyChild(1)->evaluate (x, i, gp, arg0, arg1);
                ret=gp.NthMyGene(1)->evaluate (x, i, gp, arg0, arg1);
                break;
            default:
                GPExitSystem ("MyGene::evaluate", "Undefined function value");
      }
if (isTerminal ())
  switch (node->value ())
    {
      case TERMINALX1:
            ret=x.x1[i];//Vout << "x1 op " << i << " is "  << x.x1[i]<<endl;
            break;
      case TERMINALX2:
            ret=x.x2[i]; //Vout << "x2 op " << i << " is " << x.x2[i] <<endl;
            break;
      case TERMINALX3:
            ret=x.x3[i];
            break;
      case TERMINALADF1:
            ret=arg0;
            break;
      case TERMINALADF2:
            ret=arg1;
            break;
      default:
            // This is for the constant-terminal
            ret=node->value();
      }
// Restrict the return value (it may become really large, especially
// for large trees)
const double maxValue=1e6;
if (ret>maxValue)
  return maxValue;
if (ret<-maxValue)
```

```
    return -maxValue;

  return ret;
}


void CalcVx(MyGene *pa, MyGP &pb, grids Grid, std::valarray<float> &Array)
{
    //function to calculate Vx
    for(int i = 0; i<Grid.NumberOfPoints; i++)
    { Array[i] = pa -> evaluate(Grid, i, pb, 0, 0);}
}

void CalcdVx(MyGene *pa, MyGP &pb, grids Grid, std::valarray<float> &Array)
{
    //function to calculate dVx
    double dx = 1e-6;

    //disturb states
    grids grid2a, grid2b, grid1a, grid1b;

    grid2a.disturb(Grid, 2, dx);
    grid2b.disturb(Grid, 2, -dx);
    grid1a.disturb(Grid, 1, dx);
    grid1b.disturb(Grid, 1, -dx);

    for (int i=0; i<Grid.NumberOfPoints; i++)
    {
        float Vx1a, Vx2a, Vx1b, Vx2b, fx1, fx2;

        fx1 = Func1.Eval(Grid.x1[i], Grid.x2[i]);
        fx2 = Func2.Eval(Grid.x1[i], Grid.x2[i]);

        Vx2a = pa -> evaluate(grid2a, i, pb, 0, 0);
        Vx2b = pa -> evaluate(grid2b, i, pb, 0, 0);
        Vx1a = pa -> evaluate(grid1a, i, pb, 0, 0);
        Vx1b = pa -> evaluate(grid1b, i, pb, 0, 0);

        Array[i] = (Vx2a-Vx2b)/(2*dx)*fx2+(Vx1a-Vx1b)/(2*dx)*fx1;
    }
}

// Evaluate the fitness of a GP and save it into the GP class variable
// stdFitness.
void MyGP::evaluate ()
{
    using std::valarray;

    float rawfitness=0.0;
    float diff=0.0;
    float alpha=0;

    valarray<float> VxCoarse(CoarseGrid.NumberOfPoints);
    valarray<float> dVxCoarse(CoarseGrid.NumberOfPoints);
```

```
valarray<float> VxFine(FineGrid.NumberOfPoints);
valarray<float> dVxFine(FineGrid.NumberOfPoints);

//Calculate Vx and dVx for CoarseGrid
CalcVx(NthMyGene(0), *this, CoarseGrid, VxCoarse);
CalcdVx(NthMyGene(0), *this, CoarseGrid, dVxCoarse);

// Give penalty if the function exceeds a certain length
double len=length();
if (len>cfgl.MaxLength) rawfitness+=len;

// The evaluation function checks with values of the circle-grid
for (int i=0; i<CoarseGrid.NumberOfPoints; i++)
{
    // Give penalty when V(x) <= 0
    if (VxCoarse[i]<=0) rawfitness+=1;

    //Give penalty when Vdot(x) >= 0
    if (dVxCoarse[i] >= 0) rawfitness+=1;
}

float Vmin = VxCoarse.min();
float Vmax = VxCoarse.max();
if (Vmax == Vmin && Vmax ==0) Vmax = 10;

int GridBound = CoarseGrid.NumberOfPoints*0.5;

//If rawfitness < GridBound, then calculate fitness on finer grid!
if (rawfitness < GridBound)
{
    //Calculate Vx and dVx for CoarseGrid
    CalcVx(NthMyGene(0), *this, FineGrid, VxFine);
    CalcdVx(NthMyGene(0), *this, FineGrid, dVxFine);

    for (int i=0; i<FineGrid.NumberOfPoints; i++)
    {   // Give penalty when V(x) < V0
        if (VxFine[i]<=0) rawfitness+=1;

        //Give penalty when Vdot(x) > 0
        if (dVxFine[i] >= 0) rawfitness+=1;

        //Give penalty for fixed decay rate
        if (cfgl.DecayRate && dVxFine[i] > (VxFine[i]*-cfgl.ValDecayRate)) rawfitness+=1;
    }
}

// If the rawfitness did not meet the GridBound condition for evaluation
// on the FineGrid, is has not been evaluated on the Fine Grid. It then
// could be possible that a better function (i.e. a function that did meet the
// GridBound condition) has a worse rawfitness because is got penalized on the
// FineGrid. To avoid this, we raise the rawfitness of the functions not evaluated
// on the FineGrid.
else rawfitness = rawfitness+2*(FineGrid.NumberOfPoints+GridBound);
```

```
/*ofstream test("test.txt", ios::app);
strstream strV;
strV << *this << ends;
test << "\n-----------------------------------------\n";
test << strV.str() << endl;

for (i=0; i<FineGrid.NumberOfPoints; i++) test << VxFine[i] << "\n";
test << "\n\n";
for (i=0; i<FineGrid.NumberOfPoints; i++) test << dVxFine[i] << "\n";

test.close();*/


int flag = 0;

// If fitness is zero the Lyapunov function met all the conditions. When we
// are maximizing the region of attraction or the decayrate, it is now
// time to modify those variables.
if (rawfitness == 0)
{
    // Ensuring that the Lyapunov function has closed curves
    if (cfgl.VMinMax)
    {
        // Calculate the difference between Vmin and Vmax.
        float reldif = Vmax-Vmin;

        //Give penalty for large differences between Vmax en Vmin
        if ((cfgl.VMinMax) && (reldif>=cfgl.MaxRelDif))
        {   rawfitness+=CoarseGrid.NumberOfPoints*reldif;}
    }

    //Algorithm for maximizing the decay rate or ROA.
    int ROAfitness = 0;

    //Calculate DecayRate and minimal value of DecayRate
    valarray<float> AlphaArray = -dVxFine/VxFine;
    float amin = AlphaArray.min();

    //calculate ROA and/or Decay Rate
    for (int i=0; i<FineGrid.NumberOfPoints; i++)
    {
        if (cfgl.MaxROA && VxFine[i] > Vmin) ROAfitness+=1;
        if (cfgl.MaxDecay || cfgl.DecayRate)
            { if (dVxFine[i] > (VxFine[i]*-cfgl.ValDecayRate)) rawfitness += 1;}
    }

    if (cfgl.MaxROA && ROAfitness < minFit)
    {
        minFit = ROAfitness;
        flag = 1;
        cout << "Found one - ROA.\n";
        SaveToMFile(MFile, " ", gen, "genROA", DATAINT);
    }
    rawfitness+=ROAfitness;
```

```
        if (cfgl.MaxDecay && amin > cfgl.ValDecayRate)
        {
            flag = 1;
            cout << "Found one - Decay Rate: " << amin << ".\n";
            cfgl.ValDecayRate = amin*1.25;
            SaveToMFile(MFile, " ", amin, "ValDecayRate", DATADOUBLE);
            SaveToMFile(MFile, " ", gen, "genDecay", DATAINT);
            for (int k=0;k<FineGrid.NumberOfPoints;k++)
            {   if (AlphaArray[k] < cfgl.ValDecayRate) rawfitness +=1;}
        }
    }
    else
    {
        //compensation for functions not running through rawfitness == zero loop
        rawfitness += FineGrid.NumberOfPoints;
    }

    //Save this function to the MATfile
    if (flag == 1)
    {
        strstream strVx;
        strVx << *this << ends;

        char *Vx = strVx.str();
        SaveToMFile(MFile, Vx, 0, "Vx", DATASTRING);
        OutFlag = 1;
    }

    // Do NOT forget this. The fitness is saved with the function in stdFitness and
    // NOT in the variable rawfitness.
    // We could perform some standardizing method here.
    stdFitness=rawfitness;
}

// Create function and terminal set
void createNodeSet (GPAdfNodeSet& adfNs)
{
    // Reserve space for the node set. If you are also using ADF then reserve
    // more space!!
    adfNs.reserveSpace (1);

    // Count the number of functions we are going to use during the run.
    int NumFunc = cfgl.NumberOfStates+useMUL+useSUB+useADD+
                                useDIV+usePOW+useCOS+useSIN+10*useCON;

    // Declare space for the GPNodeSet, based on the total number
    // terminals and functions we are going to use.
    GPNodeSet& ns0=*new GPNodeSet (NumFunc);

    adfNs.put (0, ns0);

    // Use this if you also need to define an ADF set:
    // GPNodeSet& ns1=*new GPNodeSet (4);
    // adfNs.put (1, ns1);

    // Define functions/terminals and place them into the appropriate
    // sets. Terminals take two arguments, functions three (the third
    // parameter is the number of arguments the function has)

    // Define the function nodes
    if (useADD) ns0.putNode (*new GPNode ('+', "+", 2));
    if (useSUB) ns0.putNode (*new GPNode ('-', "-", 2));
    if (useMUL) ns0.putNode (*new GPNode ('*', "*", 2));
    if (useDIV) ns0.putNode (*new GPNode ('%', "%", 2));
    if (usePOW) ns0.putNode (*new GPNode ('^', "^2",1));
    if (useSIN) ns0.putNode (*new GPNode ('s', "sin",1));
    if (useCOS) ns0.putNode (*new GPNode ('c', "cos",1));
    //ns0.putNode (*new GPNode ('A', "ADF0", 2));

    // Define the terminal nodes
    if (useCON) {
        ns0.putNode (*new GPNode (1, "1"));
        ns0.putNode (*new GPNode (2, "2"));
        ns0.putNode (*new GPNode (3, "3"));
        ns0.putNode (*new GPNode (4, "4"));
        ns0.putNode (*new GPNode (5, "5"));
        ns0.putNode (*new GPNode (6, "6"));
        ns0.putNode (*new GPNode (7, "7"));
        ns0.putNode (*new GPNode (8, "8"));
        ns0.putNode (*new GPNode (9, "9"));
        ns0.putNode (*new GPNode (10, "10"));
    }

    switch (cfgl.NumberOfStates)
    {
    case 3: ns0.putNode (*new GPNode (TERMINALX3, "x3"));
    case 2: ns0.putNode (*new GPNode (TERMINALX2, "x2"));
    case 1: ns0.putNode (*new GPNode (TERMINALX1, "x1"));
    }


    // Use for ADF:
    // ns1.putNode (*new GPNode ('+', "+", 2));
    // ns1.putNode (*new GPNode ('*', "*", 2));
    // ns1.putNode (*new GPNode (TERMINALADF1, "x1"));
    // ns1.putNode (*new GPNode (TERMINALADF2, "x2"));
}

// This function is not used in this implementation. Should be, because
// when reaching large numbers of generations, the program tends to jam
// in a not so neat way. Using this function should prevent that.
void newHandler ()
{
  cerr << "\nFatal error: Out of memory." << endl;
  exit (1); //Call functions registered by atexit and _onexit,
            //flush all buffers, close all open files, and terminate process
}
```

```
// Declarations of variables used for printing the results!
// Definitions are in printfile.cpp
extern ofstream tout;
extern int printTexStyle;

void main()
{
    // We set up a new-handler, because we might need a lot of memory,
    //set_new_handler(newHandler);

    // Declare the GP Variables, set defaults and read configuration
    // file.  The defaults will be overwritten by the configuration file
    // when read.
    GPConfiguration config (cout, MFile, configArray);

    minFit = 10*FineGrid.NumberOfPoints;

    // Initialize GP system.
    GPInit (0, -1);

    // Open the main output file for data and statistics file. First set
    // up names for data file. We use also a TeX-file where the
    // equations are written to in TeX-style. Very nice to look at!
    // Remember we should delete the string from the stream, well just a
    // few bytes

    int error1 = Func1.Parse(dx1, 2, 'x', 'y');
    int error2 = Func2.Parse(dx2, 2, 'x', 'y');

    if (error1 != -1 || error2 != -1)
    {
        cout << "Error parsing functions!\n";
        exit(1);
    }

    ostrstream strOutFile, strStatFile, strTeXFile;
    strOutFile  << cfgl.InfoFileDir << cfgl.InfoFileName << ".dat" << ends;
    strStatFile << cfgl.InfoFileDir << cfgl.InfoFileName << ".stc" << ends;
    strTeXFile  << cfgl.InfoFileDir << cfgl.InfoFileName << ".tex" << ends;

    ofstream fout (strOutFile.str());
    ofstream bout (strStatFile.str());
    tout.open (strTeXFile.str(), ios::out);
    tout << endl
       << "\\documentclass[10pt, a4]{article}" << endl

       //make LateX title
       << "\\title{" << cfgl.InfoTitle << "}" << endl
       << "\\date{\\today}" << endl

       << "\\begin{document}" << endl
       << "\\maketitle" << endl;
```

```
strstream txtout;
// Print the configuration to the files just opened
fout << cfg << endl;
cout << config << "\n";
tout << "\\begin{verba}\n" << cfg << cfgl << "\\end{verba}\n" << endl;

// Start the clock
clock_t start,finish;
start = clock();

// Create the adf function/terminal set and print it out.
GPAdfNodeSet adfNs;
createNodeSet (adfNs);
cout << adfNs << endl;
fout << adfNs << endl;
tout << adfNs << endl;

// Create a population with this configuration
cout << "Hit any key to abort during run.\n\n";
cout << "Creating initial population ... ";

MyPopulation* pop=new MyPopulation (cfg, adfNs);
pop->create ();
cout << "Ok.\n";
pop->createGenerationReport (1, 0, fout, bout);

// Print the best of generation to the LaTeX-file.
printTexStyle=1;
tout << *pop->NthGP (pop->bestOfPopulation);
printTexStyle=0;

// This next for statement is the actual genetic programming system
// which is in essence just repeated reproduction and crossover loop
// through all the generations .....
MyPopulation* newPop=NULL;
double fit=1000.0;
gen = 1;

// If we are maximizing the Region of Attraction or the decay rate, we don't want the
// run to stop if fitness 0 is found. Otherwise we will stop the run if fitness 0 is
// found.
while (((cfgl.MaxDecay||cfgl.MaxROA)?1:fit>0)&&gen<cfg.NumberOfGenerations&&!_kbhit())
{   // Create a new generation from the old one by applying the
    // genetic operators
    if (!cfg.SteadyState)
        newPop=new MyPopulation (cfg, adfNs);
    pop->generate (*newPop);

    // Delete the old generation and make the new the old one
    if (!cfg.SteadyState)
    {   delete pop;
        pop=newPop;
    }
    //Get fitness of Best of Population.
    fit =pop->NthGP (pop->bestOfPopulation)->getFitness();

    // Create a report of every generation and how well it is doing
    pop->createGenerationReport (0, gen, fout, bout);

    // Print the best of generation to the LaTeX-file.
    printTexStyle=1;
    tout << "Generation " << gen << ", fitness "
        << (double) pop->NthGP (pop->bestOfPopulation)->getFitness()
        << endl;
    tout << *pop->NthGP (pop->bestOfPopulation);
    printTexStyle=0;

    cout << "Generation " << gen << ", Fitness "
         << pop->NthGP (pop->bestOfPopulation)->getFitness() << endl;

    gen++;
}

// Stop the clock, calculate the elapsed time, and save it to the MATfile
finish = clock();
double duration = (double) (finish - start)/CLOCKS_PER_SEC;
SaveToMFile(MFile, "test", duration, "ElapsedTime", DATADOUBLE);
SaveToMFile(MFile, " ", gen-1, "Gen", DATAINT);

// TeX-file: end of document
printTexStyle=1;
tout << "Generation " << gen << ", fitness "
    << pop->NthGP (pop->bestOfPopulation)->getFitness()
    << endl;
tout << *pop->NthGP (pop->bestOfPopulation);
printTexStyle=0;

tout << endl
        << "\\end{document}"
        << endl;
tout.close ();

cout << "\nBest of run: " << endl;
cout << "Generation " << gen << ", Fitness "
     << pop->NthGP (pop->bestOfPopulation)->getFitness() << endl;

cout << "\nResults are in "  << cfgl.InfoFileDir
     << cfgl.InfoFileName << ".dat, "
     << cfgl.InfoFileName << ".tex, "
     << cfgl.InfoFileName << ".stc." << ends;

// Create stream object
strstream strBest;

// Stream best of Pop to object
strBest << *pop->NthGP (pop->bestOfPopulation) << ends;
```

```
    // Convert stream to char and save to MATfile.
    char *buffer = strBest.str();
    if (!OutFlag) SaveToMFile("fun.mat", buffer, 0.0, "Vx", DATASTRING);

    // Delete some stuff.
    delete[] buffer;
    delete pop;
}
```

Appendix B

The Lyapunov Class

The Interface: LyapVars.cpp

```
class LyapVariables
{ public:
  LyapVariables ();
  ~LyapVariables () {};

  int MaxLength,
      NumberOfStates,
      NumberOfPointsCircle,
      NumberOfPointsGrid,
      TestGrid,
      TestStability,
      VMinMax,
      MaxROA,
      DecayRate;

  double MaxRelDif,
      ValDecayRate;

  float RadiusCircle,
        RadiusGrid,
        PercentROA;

  char *InfoFileName;
  char *InfoFileDir;
  char *InfoTitle;
  char *fun1;
  char *fun2;

  friend ostream &operator<<(ostream& os, LyapVariables);
};
```

The Implementation: LyapVars.cpp

```
#include <stdlib.h>
#include <fstream.h>
```

```
#include "lyapvars.h"
#include "grids.h"

extern grids FineGrid, CoarseGrid;


// Setup default values
LyapVariables :: LyapVariables()
{
  NumberOfStates = 2;
  MaxLength = 15;
  MaxRelDif = 1;
  MaxROA = 0;
}

// Write variables to ostream
ostream &operator<<(ostream& os, LyapVariables lv)
{
    os << "\nLyapunov Variables                  "
     << "\nNumber of States                 = " << lv.NumberOfStates
//     << "\nNumber of points on Coarse grid  = " << lv.NumberOfPointsCoarse
//     << "\nRadius Coarse grid               = " << lv.RadiusCoarse
//     << "\nNumber of points on Fine grid    = " << lv.NumberOfPointsFine
//     << "\nRadius Fine grid                 = " << lv.RadiusFine
     << "\nVMin-Max Rule                    = " << (lv.VMinMax?"Yes":"No")
     << "\nDecay Rate                       = " << (lv.DecayRate?"Yes":"No")
     << "\nMaximize Region of Attraction    = " << (lv.MaxROA?"Yes":"No");

     os << "\n\nSaved file                          = " << lv.InfoFileDir << lv.InfoFileName
     << endl;

  return os;
}
```

Appendix C

Grids Class

The interface: grids.h

```
#include <ostream.h>
#include <math.h>
#include "mat.h"

#define PI 3.1416

class grids {
    public:  grids();
             grids(const mxArray *);

             grids &operator=(const grids &);

             void declare(const mxArray *);
             grids& disturb(grids, int, float);

             double *x1,*x3,*x2;
             int NumberOfPoints;

             void makeMT();

    friend ostream &operator<<(ostream &, grids);

    private: int NumberOfStates;
             grids &create(int, int);

};
```

The implentation: grids.cpp

```
#include "grids.h"
#include <iomanip.h>
#include <fstream.h>
#include "mat.h"
#include "mex.h"

grids::grids()
```

```
{
    NumberOfStates = 0;
    NumberOfPoints = 0;
}

grids::grids(const mxArray *pa1)
{
    declare(pa1);
}

void grids::declare(const mxArray *pa1)
{
    double *pa_x;
    int i = 0;

    if(NumberOfStates!=0)
    {
        makeMT();
        NumberOfStates = 0;
        declare(pa1);
    }

    //count number of points on grid
    NumberOfStates = mxGetN(pa1);
    NumberOfPoints = mxGetM(pa1);

    //get pointer to mxArray
    pa_x =  mxGetPr(pa1);

    //create and fill vectors
    switch (NumberOfStates)
    {
    case 3:
        x3 = new double[NumberOfPoints];
        for (i=0;i<NumberOfPoints;i++) x3[i] = pa_x[NumberOfPoints*2+i];
    case 2:
        x2 = new double[NumberOfPoints];
        for (i=0;i<NumberOfPoints;i++) x2[i] = pa_x[NumberOfPoints+i];
    case 1:
        x1 = new double[NumberOfPoints];
        for (i=0;i<NumberOfPoints;i++) x1[i] = pa_x[i];
    }
}


void grids::makeMT()    //free memory
{
    switch (NumberOfStates)
    {
        case 3: delete[] x3;
        case 2: delete[] x2;
        case 1: delete[] x1;
    }
}
```

```
//assignment-operator
grids &grids::operator=(const grids &s)
{
    if (this!=&s)
    {
        NumberOfStates = s.NumberOfStates;
        NumberOfPoints = s.NumberOfPoints;

        this->create(NumberOfPoints, NumberOfStates);

        for (int i=0;i<NumberOfPoints;i++)
        {
            switch (NumberOfStates)
            {
            case 3: this->x3[i] = s.x3[i];
            case 2: this->x2[i] = s.x2[i];
            case 1: this->x1[i] = s.x1[i];
            }
        }

    }

    return *this;
}

grids &grids::create(int NumP, int NumS)
{
    //Create a new grid
    this->NumberOfPoints = NumP;
    this->NumberOfStates = NumS;

    switch (NumberOfStates)
    {
        case 3: x3 = new double[NumberOfPoints];
        case 2: x2 = new double[NumberOfPoints];
        case 1: x1 = new double[NumberOfPoints];
    }

    return *this;
}



grids& grids::disturb(grids st, int s, float k)     //changes values of grids s
{
    this->create(st.NumberOfPoints, st.NumberOfStates);

    for (int i=0;i<NumberOfPoints;i++)
    {
        switch (NumberOfStates)
        {
            case 3: x3[i] = st.x3[i];
            case 2: x2[i] = st.x2[i];
            case 1: x1[i] = st.x1[i];
        }
        switch (s)
        {
            case 3: x3[i] = st.x3[i]+k; break;
            case 2: x2[i] = st.x2[i]+k; break;
            case 1: x1[i] = st.x1[i]+k; break;
        }
    }

    return *this;
}

ostream &operator<<(ostream &ct, grids st)
{   switch(st.NumberOfStates)
   { case 3: ct << setw(15) << "x3";
     case 2: ct << setw(15) << "x2";
     case 1: ct << setw(15) << "x1\n\n";
   }
   for (int i=0;i<st.NumberOfPoints;i++)
   {  switch(st.NumberOfStates)
    {   case 3: ct << setw(15) << st.x3[i];
        case 2: ct << setw(15) << st.x2[i];
        case 1: ct << setw(15) << st.x1[i];
      }         ct << endl;
   }
   return ct;
}
```

Appendix D

Printing

The implementation: printfile.cpp

```
#include <iostream>
#include <fstream.h>
#include <strstrea.h>

#include <stdlib.h>
#include <new.h>    // For the new-handler
#include <math.h>   // fabs()
#include <string.h>
//using namespace std;

// Include header file of genetic programming system.
#include "gp.h"
#include "gpconfig.h"

#include "symbreg.h"

// The TeX-file
ofstream tout;
int printTexStyle=0;

// Print out a gene in typical math style. Don't be confused, I don't
// make a difference whether this gene is the main program or an ADF,
// I assume the internal structure is correct.
void MyGene::printMathStyle (ostream& os, int lastPrecedence)
{
  int precedence;

  // Function or terminal?
  if (isFunction ())
    {
      // Determine operator priority
      switch (node->value ())
    {
    case '*':
    case '%':
    case '^':
```

```
  precedence=1;
  break;
case '+':
case '-':
case 's':
case 'c':
  precedence=0;
  break;
case 'A':
  precedence=2;
  break;
default:
  GPExitSystem ("MyGene::printMathStyle",
        "Undefined function value");
}

  // Do we need brackets?
  if (lastPrecedence>precedence)
os << "(";

  // Print out the operator and the parameters
  switch (node->value ())
{
case '*':
  NthMyChild(0)->printMathStyle (os, precedence);
  os << "*";
  NthMyChild(1)->printMathStyle (os, precedence);
  break;
case '+':
  NthMyChild(0)->printMathStyle (os, precedence);
  os << "+";
  NthMyChild(1)->printMathStyle (os, precedence);
  break;
case '-':
  NthMyChild(0)->printMathStyle (os, precedence);
  os << "-";
  NthMyChild(1)->printMathStyle (os, precedence);
  break;
case '^':
    os << "(";
    NthMyChild(0)->printMathStyle (os, precedence);
    os << ")^2";
    break;
case '%':
  NthMyChild(0)->printMathStyle (os, precedence);
  os << "/";
  NthMyChild(1)->printMathStyle (os, precedence);
  break;
case 's':
    os << "sin(";
    NthMyChild(0)->printMathStyle (os, precedence);
    os << ")";
    break;
case 'c':
        os << "cos(";
        NthMyChild(0)->printMathStyle (os, precedence);
        os << ")";
        break;
    case 'A':
      // This is the ADF0-function. We put the parameters in
      // brackets and start again with precedence 0.
      os << "ADF0 (";
      NthMyChild(0)->printMathStyle (os, 0);
      os << ",";
      NthMyChild(1)->printMathStyle (os, 0);
      os << ")";
      break;
    default:
      GPExitSystem ("MyGene::printMathStyle",
            "Undefined function value");
    }

      // Do we need brackets?
      if (lastPrecedence>precedence)
    os << ")";
    }

  // Print the terminal
 // if (isTerminal ())
 //   os << *node;

   if (isTerminal ())
    switch (node->value ())
      {
      case TERMINALX1:
            os << "x1";
            break;
//      case TERMINALX2:
//          os << "x_2";
//         break;
      default:
        os<< *node;
        }

}


// Print out a gene in LaTeX-style. Don't be confused, I don't make a
// difference whether this gene is the main program or an ADF, I
// assume the internal structure is correct.
void MyGene::printTeXStyle (ostream& os, int lastPrecedence)
{
  int precedence=0;

  // Function or terminal?
  if (isFunction ())
    {
```

```
  // Determine operator priority
  switch (node->value())
{
case '*':
case '%':
case '^':
  precedence=2;
  break;
case '+':
case '-':
case 's':
case 'c':
  precedence=1;
  break;
case 'A':
  precedence=3;
  break;
default:
  GPExitSystem ("MyGene::printTeXStyle",
        "Undefined function value");
}

  // Do we need brackets?
  if (lastPrecedence>precedence)
os << "\\left(";

  // Print out the operator and the parameters
  switch (node->value())
{
case '*':
  NthMyChild(0)->printTeXStyle (os, precedence);
  os << " ";
  NthMyChild(1)->printTeXStyle (os, precedence);
  break;
case '+':
  NthMyChild(0)->printTeXStyle (os, precedence);
  os << "+";
  NthMyChild(1)->printTeXStyle (os, precedence);
  break;
case '-':
  NthMyChild(0)->printTeXStyle (os, precedence);
  os << "-";
  NthMyChild(1)->printTeXStyle (os, precedence);
  break;
case '%':
  // As we use \frac, we start again with precedence 0
  os << "\\frac{";
  NthMyChild(0)->printTeXStyle (os, 0);
  os << "}{";
  NthMyChild(1)->printTeXStyle (os, 0);
  os << "}";
  break;
case '^':
    NthMyChild(0)->printMathStyle (os, precedence);
```

```
        os << "^2";
        break;
    case 's':
      os << "sin(";
        NthMyChild(0)->printMathStyle (os, precedence);
        os << ")";
        break;
    case 'c':
        os << "cos(";
        NthMyChild(0)->printMathStyle (os, precedence);
        os << ")";
        break;
    case 'A':
      // This is the ADF0-function. We put the parameters in
      // brackets and start again with precedence 0.
      os << "f_2(";
      NthMyChild(0)->printTeXStyle (os, 0);
      os << ",";
      NthMyChild(1)->printTeXStyle (os, 0);
      os << ")";
      break;
    default:
      GPExitSystem ("MyGene::printTeXStyle",
            "Undefined function value");
    }

      // Do we need brackets?
      if (lastPrecedence>precedence)
    os << "\\right)";
    }

  // We can't let the terminal print itself, because we want to modify
  // it a little bit
  if (isTerminal ())
    switch (node->value ())
      {
      case TERMINALX1:
            os << "x_1";
            break;
      case TERMINALX2:
        os << "x_2";
         break;
      case TERMINALADF1:
            os << "x_1";
            break;
      case TERMINALADF2:
            os << "x_2";
            break;
      default:
        os << node->value();
    //GPExitSystem ("MyGene::printTeXStyle","Undefined terminal value");
      }
}
```

```
// Print a Gene.
void MyGene::printOn (ostream& os)
{
  if (printTexStyle)
    printTeXStyle (os);
  else
    printMathStyle (os);
}



// Print a GP. If we want a LaTeX-output, we must provide for the
// equation environment, otherwise we simply call the print function
// of our base class.
void MyGP::printOn (ostream& os)
{
  // If we use LaTeX-style, we provide here for the right equation
  // overhead used for LaTeX.
  if (printTexStyle)
    {
      tout << "\\begin{eqnarray}" << endl;

      // Print all ADF's, if any
      GPGene* current;
      for (int n=0; n<containerSize(); n++)
    {
      if (n!=0)
            os << "\\end{eqnarray}\\\\\\begin{eqnarray}" << endl;  //for SWP
            //os << "\\\\" << endl;  //for WinEdt
          os << "f_" << (n+1) << "(x_1,x_2) & = & ";
          if ((current=NthGene (n)))
            os << *current;
      else
        os << " NONE";
        os << "\\nonumber ";
    }
     tout << "\\end{eqnarray}" << endl << endl;

    }
  else
    // Just call the print function of our base class to do the
    // standard job.
    GP::printOn (os);
}
```

Appendix E

Configuration file

The implementation: config.cpp

```
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "gpconfig.h"

#include "mex.h"
#include "mat.h"
#include "grids.h"


GPConfiguration::GPConfiguration (ostream &out, char *MFile,
                  struct GPConfigVarInformation cfg[])
  // Constructor: Read in MAT-configuration file.
{
    MATFile *pmat;
    //int MaxNumSet;

    pmat = matOpen(MFile, "r");     //open matfile
    if(pmat == NULL) cout << "Error reading functions-file\n";

// We save a pointer to the describing structure, if we need it
// later in other routines. This is quick&dirty, but well in this
// case I say use your own code if you think so
  saveStruc=cfg;


// Search for variable name in configuration array. Don't report
// an error, if not found there.  This is probably done by
// purpose.
    //int found=0;
    for (int i=0; cfg[i].varPtr!=NULL; i++)
    {
        int found = 0;
        const char *name = cfg[i].name;
        mxArray *pa = matGetArray(pmat, name);
        if (pa==NULL)
        {
            found = 0;
            cout << "Parameter " << cfg[i].name << " was not found\n";
        }
        else found = 1;

        if (found == 1)
        {
            double val = *mxGetPr(pa);
            switch(cfg[i].typ)
            {
            case DATAINT:    *(int *)cfg[i].varPtr = (int) val; break;
            case DATAFLOAT: *(float *)cfg[i].varPtr = (float) val; break;
            case DATADOUBLE:*(double *)cfg[i].varPtr = val; break;
            case DATASTRING:
                {
                    if ( mxIsChar(pa) != 1) cout << "Input must be a string.\n";

                    int buflen = (mxGetM(pa) * mxGetN(pa)) + 1;
                    char *input_buf = new char[buflen];

                    mxGetString(pa, input_buf, buflen);

                    char &input_buf2 = *new char[buflen];
                    //input_buf2 = &input_buf;
                    strcpy(&input_buf2, input_buf);

                    *(char **)cfg[i].varPtr = &input_buf2;

                    delete input_buf;
                    break;
                }
            case DATAGRID:
                {
                    grids &grid1 = *new grids;
                    grid1.declare(pa);
                    *(grids*)cfg[i].varPtr=grid1;

                    grid1.makeMT();
                }
            }
        }

        mxDestroyArray(pa);

    }

    if (matClose(pmat)!= 0 ) cout << "Error closing functions-file!\n";

}
```

```
GPConfiguration::~GPConfiguration ()
  // Destructor
{
}



void GPConfiguration::printOn (ostream& o) const
  // Print all configuration variables to o
{
  o << "# Configuration (default values), created by class configuration)\n";

  for (int i=0; saveStruc[i].varPtr!=NULL; i++)
    {
      o << saveStruc[i].name;
      for (int j=strlen (saveStruc[i].name); j<31; j++)
    o << ' ';
      o << " = ";
      switch (saveStruc[i].typ)
    {
    case DATAFLOAT:
      o << *(float *)saveStruc[i].varPtr;
      break;
    case DATADOUBLE:
      o << *(double *)saveStruc[i].varPtr;
      break;
    case DATAINT:
      o << *(int *)saveStruc[i].varPtr;
      break;
    case DATASTRING:
      o << *(char **)saveStruc[i].varPtr;
      break;
    default:
      o << "Unknown data type in internal structure\n";
    }
      o << endl;
    }
}
```

Appendix F

The Parser

Function parser for C++ v1.3 by Warp.

What's new

What's new in v1.3: (Thanks to Roland Schmehl for these bug reports).

- The library parsed wrongly sinh, cosh and tanh (confused them with sin, cos and tan and than reported a syntax error for the 'h').
- The library parsed an expression like "-cos(x)+cos(y)" like if it was "-(cos(x)+cos(y))" instead of "(-cos(x))+cos(y)" as it should. Fixed.
- The library didn't parse correctly numbers in the form "1e-2". Fixed.
- Added some explanations at the end of the file.

What's new in v1.21:

- Fixed several memory leaks (thanks to Stephen Agate for the bug report).

What's new in v1.2:

- If you define the identifier `NO_ASINH` (see the beginning of fparser.cc), then support for the functions asinh, acosh and atanh will be removed (they are not part of the ANSI C standard and thus not supported by most compilers).
- A tiny bug fixed.

What's new in v1.1:

- Fixed bug that made a negated function (eg. "-sin(x)") crash.

Preface

Often people need to ask some mathematical expression from the user and then evaluate values for that expression. The simplest example is a program which draws the graphic of a user-defined function on screen.
This library adds C-style function string parsing to the program. This means that you can evaluate the string `sqrt(1-x^2+y^2)` with given values of x and y.

The library is intended to be very fast. It byte-compiles the function string at parse time and interpretes this byte-code at evaluation time. The evaluation is straightforward and no recursions are done (uses stack arithmetic). Empirical tests show that it indeed is very fast (specially compared to libraries which evaluate functions by just interpreting the raw function string).

The library is made in ANSI C++.

Usage

To use the FunctionParser class, you have to include "fparser.hh". When compiling, you have to compile fparser.cc and link it to the main program. You can also make a library from the fparser.cc (see the help on your compiler to see how this is done).

The FunctionParser class has the following methods:

- `int Parse(const char* Function, unsigned Vars, ...);`

  Parses the given function (and converts it to internal format). Destroys previous function. Following calls to `Eval()` will evaluate the given function. The string pointed by 'Function' is not needed anymore after parsing.

  - Parameters:

Function:	Pointer to the function string
Vars :	Number of variables in the function
... :	List of variable names, char type. Eg. 'x'

  - Return values:
    * On success the function returns -1
    * On error the function returns an index to the string where the error was found (0 is the first character, 1 the second, etc). If the error was not a parsing error returns an index to the end of the string +1.
  - Example: `Parse(`3*x+y", 2, 'x', 'y');"

- `const char* ErrorMsg(void);`

  Returns a pointer to an error message corresponding to the error caused by Parse(). If no such error has occurred, returns 0.

- `double Eval(double x=0,...); double Eval(double* Vars);`

  Evaluates the function given to `Parse()` with the values given to `Eval()`. Each value given to `Eval()` corresponds the variables given to `Parse()`. There must be as many parameters as given to `Parse()`.

  - Parameters: List of doubles, one for each variable given to the `Parse()` function.

    Alternatively you can give a pointer to an array of doubles (it may be a little bit faster). There must be as many items in the array as number of variables given to `Parse()`
  - Return values:
    * On success returns the evaluated value of the function given to `Parse()`.
    * On error (such as division by 0) the return value is unspecified, probably 0.
  - Example:

```
Eval(1,-2.5);

double Vars[2]={1,-2.5};
Eval(Vars);
```

- `int EvalError(void);`

  Used to test if the call to `Eval()` succeeded.

  - Return values: If there was no error in the previous call to `Eval()`, returns 0 else returns a positive value as follows:

    1: division by zero
    2: sqrt error (sqrt of a negative value)
    3: log error (logarithm of a negative value)
    4: trigonometric error (asin or acos of illegal value)

  - Example program:

```
#include <iostream>
#include "fparser.hh"
using namespace std;

int main(void)
{
    FunctionParser Func;

    Func.Parse("x+y-1", 2, 'x', 'y');

    cout << "f(1,2) = " << Func.Eval(1.0, 2.0) << endl;

    double Vars[]={ 1.5, -2.5 };

    cout << "f(1.5,-2.5) = " << Func.Eval(Vars) << endl;

    return 0;
}
```

The function string

The function string understood by the class is very similar to the C-syntax. Arithmetic float expressions can be created from float literals, variables or functions using the following operators in this order of precedence:

`()`	expressions in parentheses first
`-A`	unary minus
`A^B`	exponentiation (A raised to the power B)
`A*B  A/B`	multiplication and division
`A+B  A-B`	addition and subtraction

Since the unary minus has higher precedence than any other operator, for example the following expression is valid: `x*-y`

The class supports these functions:

abs(A)	Absolute value of A. If A is negative, returns -A otherwise returns A.
acos(A)	Arc-cosine of A. Returns the angle, measured in radians, whose cosine is A.
acosh(A)	Same as acos() but for hyperbolic cosine.
asin(A)	Arc-sine of A. Returns the angle, measured in radians, whose sine is A.
asinh(A)	Same as asin() but for hyperbolic sine.
atan(A)	Arc-tangent of (A). Returns the angle, measured in radians, whose tangent is (A).
atanh(A)	Same as atan() but for hyperbolic tangent.
ceil(A)	Ceiling of A. Returns the smallest integer greater than A. Rounds up to the next higher integer.
cos(A)	Cosine of A. Returns the cosine of the angle A, where A is measured in radians.
cosh(A)	Same as cos() but for hyperbolic cosine.
exp(A)	Exponential of A. Returns the value of e raised to the power A where e is the base of the natural logarithm, i.e. the non-repeating value approximately equal to 2.71828182846.
floor(A)	Floor of A. Returns the largest integer less than A. Rounds down to the next lower integer.
log(A)	Natural logarithm of A. Returns the natural logarithm base e of the value A.
sin(A)	Sine of A. Returns the sine of the angle A, where A is measured in radians.
sinh(A)	Same as sin() but for hyperbolic sine.
sqrt(A)	Square root of A. Returns the value whose square is A.
tan(A)	Tangent of A. Returns the tangent of the angle A, where A is measured in radians.
tanh(A)	Same as tan() but for hyperbolic tangent.

Examples of function string understood by the class:

```
1+2
x-1
-sin(sqrt(x^2+y^2))
```

Contacting the author

Any comments, bug reports, etc. should be sent to warp@iki.fi

The algorithm used in the library

The whole idea behind the algorithm is to convert the regular infix format (the regular syntax for mathematical operations in most languages, like C and the input of the library) to postfix format. The postfix format is also called stack arithmetic since an expression in postfix format can be evaluated using a stack and operating with the top of the stack.

For example:

infix	postfix
`2+3`	`2 3 +`
`1+2+3`	`1 2 + 3 +`
`5*2+8/2`	`5 2 * 8 2 / +`
`(5+9)*3`	`5 9 + 3 *`

The postfix notation should be read in this way:
Let's take for example the expression: `5 2 * 8 2 / +`

- Put 5 on the stack
- Put 2 on the stack
- Multiply the two values on the top of the stack and put the result on the stack (removing the two old values)

- Put 8 on the stack
- Put 2 on the stack
- Divide the two values on the top of the stack
- Add the two values on the top of the stack (which are in this case the result of 5*2 and 8/2, that is, 10 and 4).

At the end there's only one value in the stack, and that value is the result of the expression.

Why stack arithmetic?
The last example above can give you a hint. In infix format operators have precedence and we have to use parentheses to group operations with lower precedence to be calculated before operations with higher precedence. This causes a problem when evaluating an infix expression, specially when converting it to byte code. For example in this kind of expression: `(x+1)/(y+2)` we have to calculate first the two additions before we can calculate the division. We have to also keep counting parentheses, since there can be a countless amount of nested parentheses. This usually means that you have to do some type of recursion.

The most simple and efficient way of calculating this is to convert it to postfix notation. The postfix notation has the advantage that you can make all operations in a straightforward way. You just evaluate the expression from left to right, applying each operation directly and that's it. There are no parentheses to worry about. You don't need recursion anywhere. You have to keep a stack, of course, but that's extremely easily done. Also you just operate with the top of the stack, which makes it very easy. You never have to go deeper than 2 items in the stack. And even better: Evaluating an expression in postfix format is never slower than in infix format. All the contrary; in many cases it's a lot faster. The above example could be expressed in postfix format: `x 1 + y 2 + /`.

The good thing about the postfix notation is also the fact that it can be extremely easily expressed in byte-code form. You only need a byte value for each operation, for each variable and to push a constant to the stack. Then you can interpret this byte-code straightforwardly. You just interpret it byte by byte, from the beginning to the end. You never have to go back, make loops or anything.

This is what makes byte-coded stack arithmetic so fast.

Appendix G

GECCO-2001 Article

Using Genetic Programming to find Lyapunov functions

I.A.C. Soute
Control Systems Technology Group
Fac. of Mechanical Engineering
Eindhoven University of Technology
5600 MB Eindhoven
The Netherlands
i.a.c.soute@student.tue.nl

M.J.G. van de Molengraft
Control Systems Technology Group
Fac. of Mechanical Engineering
Eindhoven University of Technology
5600 MB Eindhoven
The Netherlands
m.j.g.v.d.molengraft@tue.nl

G.Z. Angelis
Philips Centre for Industrial Technology
Mechatronics Research
Semiconductor Equipment Research
P.O. Box 218, 5600 MD Eindhoven
The Netherlands
georgo.angelis@philips.com

Abstract

In this paper Genetic Programming is used to find Lyapunov functions for (non)linear differential equations of autonomous systems. As Lyapunov functions can be difficult to find, we use GP to make the decisions concerning the form of the Lyapunov function. As an example two systems are taken to compute Lyapunov functions for. Although the GP algorithm is programmed to search for local stability, for most runs the result is a Lyapunov function that assures global stability. Further testing is necessary to demonstrate if similar results can be obtained for more complex systems.

1 INTRODUCTION

In 1892 Lyapunov introduced a way to prove stability of mechanical nonlinear systems (Lyapunov, 1892). Lyapunov's theory was based on energy considerations. If a system, linear or nonlinear, is always dissipating energy, except at the origin, then the system must eventually settle down to the equilibrium point in the origin. So the energy must be a positive definite function for all non-zero states. This theorem can easily be extended to arbitrary nonlinear systems. Faced with a set of nonlinear differential equations, the basic procedure of Lyapunov's direct method is to generate a scalar 'energy-like' function for the dynamic system and examine the variation in time of it.

The problem lies in finding such a Lyapunov function, given the fact that the inability to find a Lyapunov function does not mean that the system is unstable. Several techniques, e.g. the variable gradient method (Schultz and Gibson, 1962) and Krasovskii's theorem (Krasovskii, 1963) have been developed to find a Lyapunov function, but there is still no universally 'best' method for finding a Lyapunov function. The problem with these techniques is that before finding the Lyapunov function restrictive assumptions on the system have to be made. As will be shown, few restrictions and assumptions are necessary when trying to find a Lyapunov function using GP.

In Section 2 the Lyapunov theorem will be explained in more detail. Section 3 explains the GP algorithm. Then, in Section 4, the GP algorithm will be applied to two problems. The paper will be concluded in Section 5.

2 LYAPUNOV'S DIRECT METHOD

Consider the set of (non)linear differential equations represented as in Eq. (1)

$$\dot{\mathrm{x}} = \mathbf{f}(\mathrm{x}) \tag{1}$$

The Lyapunov theorem for local stability states that the equilibrium at the origin is locally (asymptotically) stable when two conditions are met.

Theorem 1 (Local stability) *If, in a ball $\boldsymbol{B}_{R_0}$, there exist a scalar function $V(\boldsymbol{x})$ with continuous first partial derivatives such that*

- *$V(\boldsymbol{x})$ is positive definite (locally in $\boldsymbol{B}_{R_0}$)*
- *$\dot{V}(\boldsymbol{x})$ is negative semi-definite (locally in $\boldsymbol{B}_{R_0}$)*

then the equilibrium point $\boldsymbol{0}$ is locally stable. If actually the derivative $\dot{V}(\boldsymbol{x})$ is locally negative definite in $\boldsymbol{B}_{R_0}$, then the stability is asymptotic.

$\dot{V}(\mathrm{x})$ can be calculated as follows:

$$\dot{V}(\mathrm{x}) = \frac{dV(\mathrm{x})}{dt} = \frac{\partial V}{\partial \mathrm{x}}\,\dot{\mathrm{x}} = \frac{\partial V}{\partial \mathrm{x}} f(\mathrm{x}) \tag{2}$$

To ensure global stability Lyapunov posed the following theorem:

Theorem 2 (Global stability) *Assume that there exists a scalar function V of the state $\boldsymbol{x}$, with continuous first order derivatives such that*

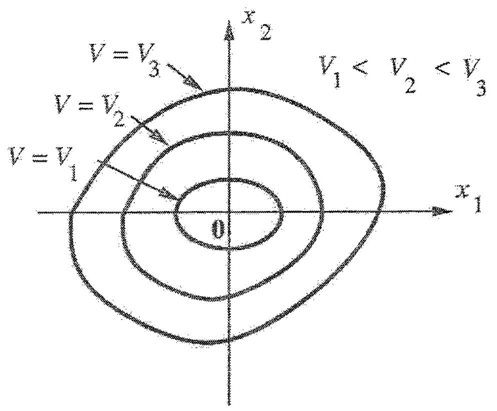


Figure 1: Graphical interpretation of a positive definite function

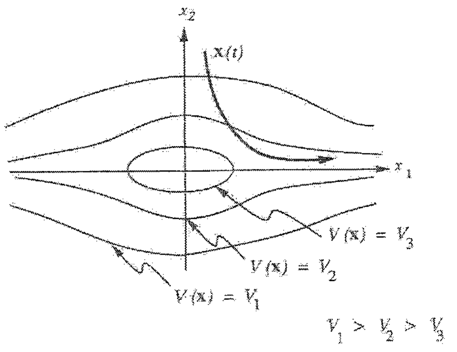


Figure 2: Example of open contour curves

- *$V(\boldsymbol{x})$ is positive definite for all $\boldsymbol{x}$*
- *$\dot{V}(\boldsymbol{x})$ is negative definite for all $\boldsymbol{x}$*
- *$V(\boldsymbol{x}) \to \infty$ as $\|\boldsymbol{x}\| \to \infty$*

then the equilibrium at the origin is globally asymptotically stable.

As an example, a graphical representation of a positive definite function for a 2-dimensional system can be made as follows: taking x_1 and x_2 as Cartesian coordinates we can draw contour curves, each corresponding to a positive value of V, see Figure 1. Furthermore, we would like the curves to be closed, at least locally around the origin. If the curves are not closed, it is possible for the state trajectories to drift away from the equilibrium point, even though the state keeps going through contours corresponding to smaller and smaller V's, see Figure 2.

3 GP ALGORITHM

We present a set of (non)linear differential equations to the GP algorithm. The GP algorithm evolves a set of Lyapunov functions candidates and assigns them a fitness value, based on how well the candidate function satisfies the conditions of the local stability theorem (Theorem 1).

The GP run is performed using a C++ program (Fraser and Weinbrenner, 1997) with modifications to allow evaluation in MATLAB (The MathWorks Inc., 1999). Calculations are performed on a 500 MHz Pentium 2 processor with 128 Mb internal memory, running Windows 95.

Before the fitness of the GP-generated Lyapunov function can be calculated, a number of fitness cases has to be defined.

3.1 CHOICE OF FITNESS CASES

To prove local stability around the origin, the Lyapunov function has to meet the criteria stated in Theorem 1 within a certain area. Points in this area are taken as fitness cases, starting with the points on the edge of this area. These points are typically taken in a circle around the origin for a 2 dimensional system (see Figure 3), and in a hyper-ball around the origin for higher dimensional systems. Also a rectangular grid is defined (see Figure 4). The circle grid is always calculated in a way that it always includes the points on the axes to assure a positive definite function containing all nonzero states. The origin is excluded from the rectangular grid because it cannot meet the Lyapunov requirements, i.e. in the origin $V(\mathbf{x}) = 0$ and $\dot{V}(\mathbf{x}) = 0$.

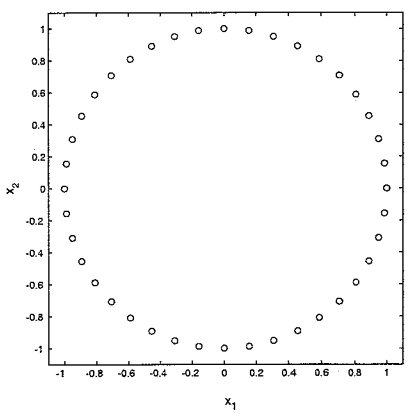


Figure 3: Circle grid

3.2 FITNESS CALCULATION

Best of run is the function with fitness value zero. The fitness of a GP-generated Lyapunov function candidate can be calculated as follows:

1. Evaluate $V(\mathbf{x})$ for all fitness cases $\mathbf{p}$. If $V(\mathbf{x}) \leq 0 \; \forall \; \mathbf{p}$ then increase the fitness value.
2. Calculate $\dot{V}(\mathbf{x})$.
3. If $\dot{V}(\mathbf{x}) \geq 0 \; \forall \; \mathbf{p}$ then increase the fitness value.

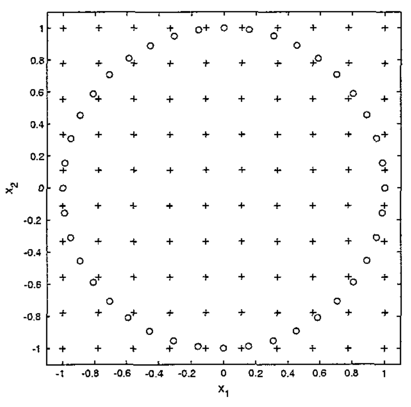


Figure 4: Circle and fine grid

4. (optional) Calculate difference between V_{min} and V_{max}. If this value is larger than α then increase the fitness value.

5. If $fitness = 0$ then calculate fitness for the points on the rectangular grid (repeating step 1 to 3).

6. If $fitness \neq 0$ then

$$fitness = fitness + np_g$$

where np_g is the number of points on the rectangular grid. Candidates that do not meet all the conditions for the circle grid, are not evaluated on the rectangular grid. Suppose that candidate A does not meet the conditions on the circular grid, it lacks one point. According to the rules described above, candidate A would be assigned fitness value 1. Suppose now that candidate B does meet all conditions for the circle grid points, but doesn't meet all the conditions for the rectangular grid, it lacks five points. In the end this would mean that candidate A is a better candidate than B, which is not true. To prevent this, the number of points on the rectangular grid is added to the fitness value of candidates that are not evaluated on the rectangular grid.

7. (optional) Standardize or normalize fitness value.

Item 4 has been added to promote that the contour curves of the Lyapunov function will be closed, as discussed in Section 2. By choosing a value for α we can control the form of the contour curves. When α is a large value, V_{min} and V_{max} are allowed to differ greatly, and therefore the forms of the contour curves do not much resemble circular forms. When the value for α is chosen zero, then V_{min} and V_{max} are forced to become equal, which will result in contour curves that approximate circles, and are therefore closed curves.

4 TESTING THE GP ALGORITHM

We have to keep in mind that the Lyapunov function found by the GP algorithm is only valid for the points tested on the grid, and not for the whole domain. So the function has to be tested analytically afterwards. One of the options to perform this test is by looking at the contour plot of $V(\mathbf{x})$, as described in Section 2. To test the GP algorithm some problems have been posed, beginning with a very simple system. The settings for the GP algorithm are the same for each problem discussed:

Population Size	100
Crossover probability	0.7
Mutation probability	0.3
Genetic Operators	+,-,×
Tournament selection	

4.1 SIMPLE PROBLEM

To test the GP algorithm we start with a very simple problem of which we know the answer in advance.

$$\begin{aligned}\dot{x}_1 &= -x_1\\ \dot{x}_2 &= -x_2\end{aligned} \tag{3}$$

A valid Lyapunov function for the system described in Eq. (3) is $V(\mathbf{x}) = x_1^2+x_2^2$. This function is positive definite for $\mathbf{x} \in \mathcal{R}^2$. Furthermore, the derivative of $V(\mathbf{x})$ is $\dot{V}(\mathbf{x}) = -2x_1^2-2x_2^2$, which is negative definite for $\mathbf{x} \in \mathcal{R}^2$.

We start the GP run with 24 fitness cases on a circle with radius 1 around the origin, and we do not enforce the V_{min}/V_{max} rule. The results are listed in Table 1.

Run	Gen.	Fitness	$V(\mathbf{x})$
1	2	0	$x_1^2+x_2^2$
2	2	0	$x_1^2+x_2^2+x_1x_2$
3	3	0	$x_1^2+x_2^2$
4	7	0	$x_1^2-2x_2^2+4x_1-x_2$
5	3	0	$x_1^2+3x_2^2+(2x_2-2)x_1$
6	3	0	$x_1^2+x_2^2+x_1x_2$
7	2	0	$x_1^2+x_2^2$
8	3	0	$x_1^2+x_2^2+x_1x_2$
9	2	0	$x_1^2+x_2^2$
10	8	0	$x_1^2+x_2^2+x_1x_2$

Table 1: Results

Only 4 out of 10 runs resulted in $V(\mathbf{x}) = x_1^2+x_2^2$ which ensures global asymptotic stability. The Lyapunov function $V(\mathbf{x}) = x_1^2+x_2^2+x_1x_2$, does have closed contour curves, but its derivative is: $\dot{V}(\mathbf{x}) = -2x_1^2-2x_2^2(1+2\frac{x_1}{x_2})$. This result is only negative definite if $(1-\frac{x_1}{x_2}) > 0$, so only local stability is ascertained. The Lyapunov functions of runs 4 and 5 are not even positive definite. The reason that these functions are found as an answer of the GP run, is that for each of the points that was tested, the conditions of theorem 1 are valid, but they are not for the whole domain.

When applying the V_{min}/V_{max} rule, 9 out of 10 runs result in $V(\mathbf{x}) = x_1^2 + x_1^2$. The last result didn't end up with fitness zero. The maximum number of generations was set to 100. At generation 100 $V(\mathbf{x})$had fitness 4. It would probably had resulted in fitness 0, if the maximum number of generations had been higher.

Although we were only looking for local stability, the end result $V(\mathbf{x}) = x_1^2+x_2^2$ meets the criteria for global stability (Theorem 2).

4.2 NONLINEAR PROBLEM

We submitted the set of nonlinear differential equations, described in Eq. (4), to the GP algorithm.

$$\begin{aligned} \dot{x}_1 &= -2x_1 \\ \dot{x}_2 &= -2x_2 + 2x_1x_2^2 \end{aligned} \qquad (4)$$

We use only the circle grid with radius 1 and 24 points. The demand for $V_{min} = V_{max}$ is in use.

Although the number of generations needed to achieve fitness zero differed, the Lyapunov functions found for the system are all the same: $V(\mathbf{x}) = x_1^2+x_2^2$. The derivative for this Lyapunov function is $\dot{V}(\mathbf{x}) = -4x_1^2 - 4x_2^2(1 - x_1x_2)$. In the region $(1 - x_1x_2) > 0$ $\dot{V}(\mathbf{x})$ is locally negative definite. As $V(\mathbf{x})$ is positive definite, the stability is (locally) asymptotic.

A similar result is deduced using the variable gradient method (see Slotine and Li, 1991). To use the variable gradient method, one has to assume a form of the Lyapunov function beforehand. Using GP to find a Lyapunov function, we do not need to do this.

5 CONCLUSIONS AND FUTURE RESEARCH

Finding a Lyapunov function and thereby proving (local) stability of the equilibria of a system is not an easy task, but GP can lighten the task a little. Although only simple systems have been investigated here, the results are promising. Admittedly, not every run ends up in a valid Lyapunov function, but one does not have to go into complex mathematical calculations to obtain a Lyapunov function. Only validation of the Lyapunov function is needed afterwards.

To assure that more runs will result in a valid Lyapunov function, the fitness evaluation has to be improved. More specifically: the fitness cases have to be chosen carefully. We need to try to choose the grid density in a way that will ensure that $\dot{V}(\mathbf{x})$ will not significantly change between two grid points. This concept has to be investigated further. The rectangular has not been used, during the search for Lyapunov functions, because the problems evaluated here could be solved using only the circle grid. We expect that, when evaluating more complex problems, the rectangular grid will be needed.

So far, only systems with two states have been tested. The program is capable of searching for Lyapunov functions for higher dimensional systems, which will be tested in the near future.

During the runs we did not experiment with the GP settings for e.g. number of generations, the population size etc. Further testing is needed on how these parameters can influence the search for a Lyapunov function. Furthermore, we did not use more than three genetic operators, thereby reducing the flexibility of the solutions. By using more operators a greater variety of possible solutions can be addressed.

Another point for future research is to compare this method of finding Lyapunov functions to other techniques. It is clear that by using GP we do not need to use difficult mathematical methods, but we still have to examine whether GP finds comparable or even better Lyapunov functions than other techniques.

The next step is to use this GP algorithm in a larger problem, namely identification of systems. In previous research we tried to dynamically identify a physical system. Using GP resulted in a mathematically correct function, but in simulation it turned out to be an unstable system. By applying the Lyapunov search during the identification process, we expect to overcome this problem.

References

Fraser, A. and Weinbrenner, T. (1997). *The Genetic Programming Kernel Version 0.5.2.*

Krasovskii, N. (1963). *Problems of the theory of stability of motion.* Stanford Univ. Press, Stanford. translation of the Russian edition, Moskow (1959).

Lyapunov, A. (1892). *The General Problem of Motion Stability.*

Schultz, D. and Gibson, J. (1962). *The Variable Gradient Method for Finding Lyapunov Functions*, volume 81 of *AIEE Trans. part II, Appl. & Industry*, pages 203–210.

Slotine, J. and Li, W. (1991). *Applied Nonlinear Control.* Prentice Hall.

The MathWorks Inc. (1999). *Using MATLAB.*