UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

# High-performance simulation of the 16th Hilbert's problem

June 21, 2021

**Aleix Boné Ribó**

Bachelor thesis
Specialization in Computing

Director: Grigori Astrakharchik

# Contents

# List of Figures

# List of Tables



# List of Listings

**Abstract**

A high performance program to find limit cycles in quadratic ordinary systems of equations (ODEs) using the computational power of GPUs (Graphical Processing Units) was developed. The program is able to find limit cycles on a quadratic system in seconds, allowing for a quick visualization of the cycles or to perform a massive parameter search to find systems with interesting limit cycle configurations.

# 1 Context and scope

## 1.1 Introduction and contextualization

Since its release in 2007, Compute Unified Device Architecture (CUDA) has revolutionized the usage of graphic processing units for scientific computations, allowing developers to implement programs that take full advantage of the parallelization capabilities of Graphics Processing Units (GPUs) for general purpose programming. This paired with the exponential growth of computing power that GPUs have experienced in the last decade has made GPU numerical analysis essential in modern science research. Highly complex problems that were once impossible to compute in realistic time frames can now be computed even on average consumer hardware GPUs. Moreover, projects like *GPUGRID*[1] allow researchers to run distributed programs through a grid of GPUs from volunteers all over the world reaching supercomputing level performance [1].

In 1900 David Hilbert posed a list of 23 important problems in the field of mathematics which were unsolved at the time [17, 16]. Those problems have been vastly studied since then and most of them are solved or partially solved. There are however some which are still unsolved to this day.

One of the still unsolved problems is the 16th problem. This problem consists of two separate problems, the first one regarding the relative positions of the branches of real algebraic curves and the second one about the upper limit of *limit cycles* on two-dimensional vector fields and their relative positions. In this project we are going to study the second part of this 16th problem.

In particular, we study the number of limit cycles for vector fields of polynomials of second degree:

$$\begin{aligned} \frac{dx}{dt} &= a_1x^2 + b_1xy + c_1y^2 + \alpha_1x + \beta_1y \\ \frac{dy}{dt} &= a_2x^2 + b_2xy + c_2y^2 + \alpha_2x + \beta_2y \end{aligned} \tag{1}$$

Solving the system of differential equations (1) results in a trajectory $x(t), y(t)$ which represents some curve on $(x, y)$ plane. There are different possibilities for how this curve looks like. One of the most interesting cases is when it results in

[1] www.gpugrid.net

an *attractor*. An attractor is defined as a compact subset of the phase space of a dynamical system, such that all trajectories from some neighborhood of it tend to it at infinite time. An attractor can be an attracting fixed point or as well a periodic trajectory (limit cycle).

- the trajectory goes away to infinity
- the trajectory goes to a fixed point
- the trajectory goes to a limit cycle

It is unknown whether there exists an upper limit on the number of limit cycles for systems with polynomials of degree greater than one [9, 19, 24]. Nowadays, the largest number of limit cycles found is equal to four. It is still an open question whether or not a larger number of limit cycles is possible. Therefore, finding a system with 4 limit cycles or more will be quite an important result for the academic community.

### 1.1.1 Context

This is a Bachelor Thesis of the Computer Engineering Degree, specialization in Computing, done in the Facultat Informàtica de Barcelona (FIB) of the Universitat Politécnica de Catalunya (UPC). The project is directed by Grigori Astrakharchik.

### 1.1.2 Concepts

Below are some concepts needed to understand the project.

#### Limit cycles

A limit cycle is a closed trajectory with the property that at least one other trajectory spirals into it as time approaches infinity. They are important in various applications in the field of dynamical systems.

#### CUDA

CUDA is a parallel computing platform developed by *nvidia* that allows general computing on their graphic processing units (GPUs). Using the CUDA programming model allows developers to run massively parallel programs on GPUs.

### 1.1.3 Problem to be solved

There have been various studies on the number of limit cycles for second degree vector fields but so far the maximum number of cycles found is 4, as reported in Ref. [22]. The aim of this project is to search the parameter space for systems that have 4 or more cycles to gain more insight on the nature of these equations.

Figure 1.1 shows a characteristic example of a visualization of the limit cycles in a two-dimensional polynomial quadratic system.

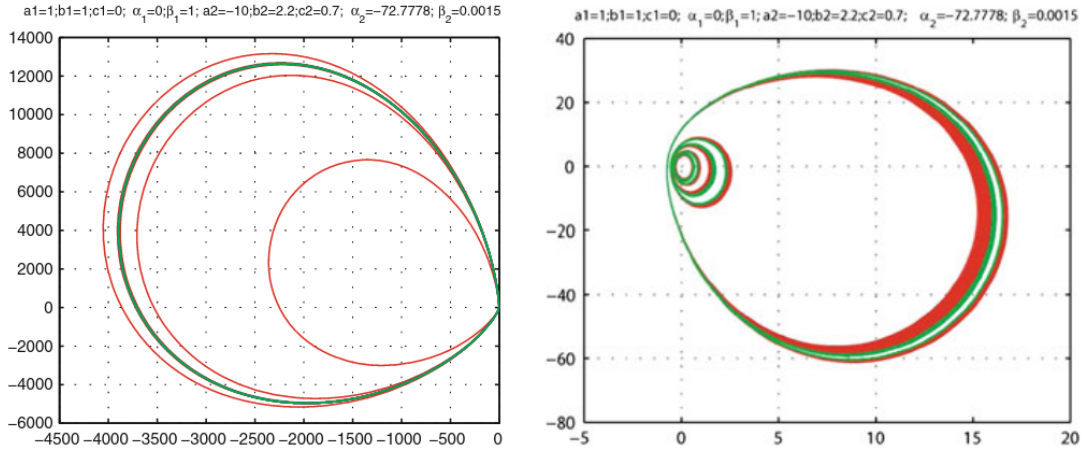


Figure 1.1: Visualization of four limit cycles in two-dimensional polynomial quadratic system, from Ref. [22]

We implement an efficient parallel algorithm that solves systems of ordinary differential equations (ODE) and detects limit cycles for a wide range of parameters and points in the plane. This code is implemented in Julia programming language and will use CUDA framework in such a way that massive parallel calculation can be done on a dedicated CUDA server with several advanced GPU graphic cards.

## 1.2 Computational complexity

The system of differential equations eq. (1) contains 5 free coefficients for $x$ and 5 more for $y$ which makes a total of 10 distinct parameters. Adding the initial point to calculate the trajectories ($x$ and $y$ coordinates) it makes a total of 12 free parameters. Thus, the total space of parameters to be investigated is huge. Fortunately, the number of coefficients in the study can be reduced further to only 5 independent ones (parameters for $x$ are 1) as given in eq. (2).

$$\begin{aligned} \frac{dx}{dt} &= x^2 + xy + y^2 + x + y \\ \frac{dy}{dt} &= a_2x^2 + b_2xy + c_2y^2 + \alpha_2x + \beta_2y \end{aligned} \tag{2}$$

This ends up in a total of 7 independent parameters.

If we consider $n$ different values for each of those parameters we have a total of $n^7$ different trajectories to calculate. For example, for $n = 100$ different values for each parameter, $10^{14}$ trajectories has to be calculated in order to sample the space of available parameters. If we were to compute the trajectories sequentially it would take an exceedingly long time even for moderately sized values of $n$. Calculating a trajectory on a typical desktop machine takes 0.01 seconds. This means that for $n = 10$ it would take approximately 28 hours to compute all the trajectories and taking $n = 30$ the time increases to 7 years. If we were to use a GPU with 5000 CUDA cores (Tesla K90) it would only take 20 seconds (assuming perfect parallelization). And for $n = 30$ it would take 12 and a half hours. Thus, the use of CUDA clusters, like Minotauro in Barcelona Supercomputing Center, looks very promising. If we consider the usage of a cluster composed of 39 servers with 2 Tesla K90 GPUs the computation time for $n = 30$ is estimated to be a mere 9 minutes. [2]

### 1.2.1 Stakeholders

The main stakeholders in this project are the researcher (me) and the director, Grigori Astrakharchik, that have a direct implication on the Thesis. If we manage to obtain interesting results other parties may benefit from them: other researchers on the fields of numerical analysis and dynamic systems may build on these results.

[2] All these calculations are based on a very rough estimate of the computing time needed to determine the limit cycle of a trajectory which may vary a lot depending on the system and the implementation of the code

## 1.3 Justification

In [22] there is a description of a task given by the academician A.N. Kolmogorov:

> To estimate the number of limit cycles of square vector fields on plane, A.N. Kolmogorov had distributed several hundreds of initial parameters (with randomly chosen coefficients of quadratic expressions) among a few hundreds of students of Mech & Math Faculty of Moscow State University as a mathematical practice. Each student had to find the number of limit cycles of a field. The result of this experiment was absolutely unexpected: not a single field had a limit cycle!

This shows that the parallelizable nature of the problem and how difficult it is to find those cycles. Therefore, it is important to implement a code that is both efficient on the calculation and have a big enough search space to find results.

### 1.3.1 Previous studies

There has been a number of studies relying on numerical methods to find limit cycles in two-dimensional vector fields [23, 18, 7, 13]. Papadimitriou and Vishnoi showed that the computation of a limit cycle is PSPACE-complete [28]. The maximal known number of limit cycles is reported in Kuznetsov et al. [22] where an example of specific conditions for which **4 limit cycles** is provided.

All the previous studies have relied on standard analytical methods to calculate trajectories and find limit cycles, there has not been research on developing efficient parallel code to solve ODE trajectories on GPUs with the aim of finding limit cycles. This projects aims to develop a highly efficient method to aid the search of limit cycles using modern computational capabilities. At best, we will be able to find systems with 4 or more limit cycles on our search, at worst the developed code will still be useful to quickly analyze systems in search of limit cycles.

### 1.3.2 CUDA and Julia

The CUDA framework has API for several programming languages (C, C++, Fortran, Python, MATLAB, …). The official programming toolkit [26] is in C/C++ and offers the most customizability and low level configuration to adapt the code to the hardware. The two most notable alternatives are Python's *pyCUDA* [21] and Julia's *CUDA.jl* [4] libraries. These libraries bind to the C CUDA API and interface the data between the kernels and the programming language. As such, the performance of the kernels that run in the GPU should be equivalent in all cases but the data models and processing of different languages makes a difference when interfacing between the GPU code and the CPU code.

However, languages such as Python and Julia allow for faster development of the code and easier visualization of the data. For this Thesis we will use Julia programming language since it's faster than Python, has many libraries and tools for numerical analysis and has type checking. The performance of Julia CUDA library has a 0.50% impact on performance compared to C according to the literature [4]. Moreover, the *DifferentialEquations.jl* Julia library [32] has been shown to outperform C implementations of ODE solvers and currently is the most feature complete and efficient suite for differential equation solving [31].

## 1.4 Scope

### 1.4.1 Objectives and sub-objectives

The main objective of this Thesis is to develop a highly-efficient code capable of determining the possible existence of limit cycles for a system. This code must also be capable of being executed in parallel in a GPU cluster making full use of its computing power to simulate a wide variety of systems with different parameters. Furthermore, the results must be processed to find interesting systems to visualize and analyze in more detail. These objectives can be divided in sub-objectives:

#### Theoretical part

Before implementing the algorithms and developing the code, a deep understanding of the current numerical methods and the CUDA framework must be achieved to find the best approach to the problem.

- Explore the best strategies to solve ordinary differential equations.
- Explore numerical methods to verify the existence of limit cycles and determine the number of cycles for a given system.
- Research how these methods can be applied to run in a CUDA system efficiently.

#### Practical part

Having done the background research, the code must be developed, implemented and tested. Different methods will be tried in order to find the ones that give the best performance.

- Implement the algorithms in an efficient code.
- Benchmark the performance and compare different approaches to achieve the best performance.
- Run the code on a dedicated GPU server.
- Analyze the results and visualize them.
- Present and discuss the obtained results in the Thesis.

### 1.4.2 Requirements

To ensure the quality of the Thesis a number of requirements must be fulfilled:

- Find the best balance between accuracy of the results and computational complexity
- Ensure that the numerical methods applied are properly implemented.
- Take into account numerical stability of the methods used as well as rounding and overflow errors.
- Profile the different methods under the same conditions and environment to ensure that there are no biases.
- Use good programming practices, making readable and maintainable code with the least complexity possible.
- Make the developed code publicly available.

### 1.4.3 Potential obstacles and risks

There are several risks that may have to be dealt with during the development of this Thesis.

- **Project deadlines**: There is a limited amount of time to do the project. Therefore, a proper planning of tasks and time must be made and followed to ensure that the work can be done in the proper time frame.
- **Computational power**: This Thesis involves a lot of computational power and the whole project is conditioned by it. If the program cannot be run on the proper hardware the results may not be obtainable in a realistic time frame and the scope of the project will have to be reevaluated.
- **Inexperience on the field**: I have very limited experience with CUDA programming and just the basics of numerical computation techniques, so there is a lot of research to be done, specially regarding dynamical systems.

## 1.5 Methodology and rigor

### 1.5.1 Methodology

Since the Thesis must be completed in a relatively short period of time we will apply the *agile* methodology and divide the work into sub-tasks or *sprints*. These *sprints* will consist of different stages of implementation of the program, beginning with a proof of concept running in sequence on the CPU and progressively iterating on this base to optimize the methods and adapt them to be able to run in CUDA on the GPU.

### 1.5.2 Monitoring tools and validation

To manage the different iterations of the code `git` will be used, this will allow to have a complete history of the different changes to the code made during the project. Different branches will be used during the development of the code: a *master* branch with the current working code, a *development* branch in which the newest features are added. When the work done in the *development* branch is properly tested and proved to be working the changes will merge into the *master* branch. Additionally, some other branches may be created with experimental changes.

The `git` repository will be hosted on GitHub and access will be granted to the project tutor allowing him to follow and monitor the work and results at any time. To control the bugs and features added to the code *GitHub*'s issue system will be used, this issues will be classified with labels and managed through *GitHub*'s project system (using kanban style project boards) following *GitHub*'s guidelines on project management with their platform [14]. Every time a milestone is reached a new signed label will be added to the repository to keep track of the most relevant points in the development of the program. The repository will be private during the development of the thesis but will be made publicly available to the community once it's finished.

The thesis' document will also be tracked with `git` and hosted on *GitHub* similarly to how the code is monitored. However, the document repository will also be synced to Overleaf allowing online edition of the document.

A weekly meeting with the tutor will be arranged to discuss the progress and which tasks should be worked on.

# 2 Time planning

The work-to-begin date is on February 9th and the delivery date on June which gives around 18 weeks of time. I plan to work on the project approximately 35 hours a week, this gives around 630 hours in total working on the project.

## 2.1 Description of the tasks

### 2.1.1 Task definition

The definitions of the tasks that will be done throughout the project are divided into 4 categories: planning, research, implementation and experimentation.

#### Project planning

- **Contextualization and project scope** Describe the project scope and its context. Giving an overview of the objective, other past studies on the topic and how this project is relevant.
- **Time planning** Organize the work to be done in granular tasks to estimate the time needed for each of them. With the time estimation create a realistic planning for the tasks completions so that the project can be finished in the appropriate time frame.
- **Budget and sustainability** Analyze the economic and environmental sustainability of the project.
- **Meetings** Each week a meeting with the tutor will be scheduled to ensure that the thesis is proceeding correctly and within the expected deadlines.
- **Integration into the final document** All these project planning tasks must be integrated into the final thesis memoir.

### Research

Given the nature of this thesis it is fundamental to have a solid understanding of modern numerical computation techniques applied to solving Ordinary Differential Equations and finding limit cycles. It is also primordial to know how these techniques can be applied to CUDA and how to benchmark the programs to detect bottlenecks and find whether the processing power of the GPU is being used in its full potential.

- **Research ODE solvers** There are lots of algorithms to solve ODEs which will have to be tried to find the ones that give the best results in our case.
- **Research how to find limit cycles** Find the best approach to detect limit cycles
- **Research CUDA** This task includes reading the official documentation as well as other sources on numerical methods applied to CUDA.

### Practical Implementation

The different algorithms researched must be implemented in order to experiment with them and find the best ones to solve the problem.

- **Program different methods to find limit cycles** As stated before various different methods will have to be implemented. This implementation can be initially without using CUDA (fully sequential).
- **Adapt de code to be run with CUDA** Once the methods to find limit cycles are properly implemented they must be adapted in order to be run with CUDA.
- **Test the program** In order to ensure that the code implemented is correct and the errors introduced in the computation fall within a reasonable distance of the real theoretical value some tests must be implemented. This includes also the implementation of small programs to visualize the results and interpret them.

### Experimentation, analysis and conclusions

Once there is a working prototype of the code, the experimentation can begin. There will be two major parts of the experimentation, the first one when the various methods are tested with a relatively small number of cases to find the best one (taking into account both the speed and accuracy of the calculations) and the final experiment when the best method is run with a much bigger search space and from which the results can be analyzed.

- **Comparison experiment**
  - **Select search space** Decide how many parameters will be used on the experiment. It should be a big enough search space so as to have variety on the systems but not big enough that it takes too much time to benchmark and slows down the development.
  - **Select benchmarks** The benchmarks must be run in similar conditions, and we must decide what metrics will be considered (execution time, FLOPS[3], accuracy, memory usage, …).
  - **Further optimize the methods** When running the benchmarks we may find some possible improvements on the original implementations which can then be improved and tested again.
  - **Analyze results and decide the best method** With the results of the experiments we can decide on which method is the best and should therefore be used in the final experiment.
- **Final experiment**
  - **Select search space** Given the results of the previous experiments, estimate the runtime as a function of the search space and select a search space as big as possible given the available computing time.
  - **Analyze results** Once the final experiment finishes the results can be analyzed in search of systems with interesting number of limit cycles.
- **Conclusions** Analyze the results of both experiments and provide a conclusion.

[3]Floating Point Operations Per Second

### 2.1.2 Estimation of the number of hours per task

It is quite difficult to estimate the time needed for each of the proposed tasks since there is no clear reference of how long they can potentially take. To account for possible delays I will take an upper bound on the hours needed for each task.

The **project planning** should be fairly quick, so I estimate around 30 hours of work for the first tasks. There will be 18 meetings of 1 hour and the integration into the final document should take around 12 hours taking into account all changes and corrections that may arise.

As explained in section 2.3.3 a fair amount of time will be devoted to **research**. For each of the 3 main topics I will research around 50 hours to ensure I can get a deep understanding of the topic.

The **implementation** is the part that should take most time given that there are multiple different methods to try and test. The implementation of the methods should take around a month ($\sim$ 100 hours) and the adaptation to CUDA 2 weeks ($\sim$ 50 hours). While doing the different methods, testing should be performed this should add up to around 50 hours of testing.

Regarding the **comparison experimentation** the optimization of methods should take between 50 and 100 hours given that the initial implementation may require a lot of tweaking and fine-tuning to adapt to the hardware. The analysis of results should be fairly quick if the code produces proper metrics. In the **final experimentation** most of the time will be spent on the computing and the work should be around 20 hours of monitoring the execution and analyzing the results.

Table 2.1 shows the summary of the tasks with their time estimate and dependencies. The Gantt chart is shown in fig. 2.1. It must be noted that this time estimates are subject to change.

### 2.1.3 Summary of the tasks

Table 2.1: Summary of tasks

| ID | Task | Time (h) | Depend. |
|---|---|---|---|
| P | **Project planning** | **60** | |
| P0 | - Contextualization and project scope | 10 | |
| P1 | - Time planning | 10 | |
| P2 | - Budget and sustainability | 10 | |
| P3 | - Meetings | 18 | |
| P4 | - Integration into final document | 12 | |
| R | **Research** | **150** | |
| R0 | - ODE solvers | 50 | |
| R1 | - Limit cycles | 50 | |
| R2 | - CUDA | 50 | |
| I | **Implementation** | **200** | |
| I0 | - Different methods to find limit cycles | 100 | R0,R1 |
| I1 | - Adapt to CUDA | 50 | I0,R2 |
| I2 | - Tests | 50 | |
| EC | **Experimentation (comparison)** | **130** | I |
| EC0 | Select search space | 5 | |
| EC1 | Select benchmarks | 5 | |
| EC2 | Further optimize the methods | 100 | |
| EC3 | Analyse the results and decide the best method | 20 | |
| EF | **Experimentation (final)** | **25** | EC |
| EF0 | Select search space | 5 | |
| EF1 | Analyse the results | 20 | |
| C | **Conclusions** | **10** | EC,EF |
| D | **Documentation** | **60** | |
| O | **Oral exposition** | **10** | |

### 2.1.4 Resources

#### Human resources

The main human resource of the thesis is the researcher. There is also the director Grigori Astrakharchik which mentors the researcher and the GEP Tutor Eguiguren Huerta Marcos in charge of correcting the project management part.

#### Material resources

The main resources needed for this project are previous papers and books on the topics of ODEs, limit cycles and CUDA. For the implementation an execution of the code there following resources will be used:

- **VCS**: `git` will be used as a version control system (VCS) and the code will be hosted on *GitHub* for easy collaboration with the director.
- **LaTeX**: To format the document LaTeXwill be used. The document will be hosted on *Overleaf* to allow easier collaboration and due to the fact that it provides *github* integration.
- **Atenea**: To communicate with the GEP tutor.
- **Computers**: My own personal computer with an *RTX2060* will be used to develop the code and test it. The final versions will run on a server from the department of physics at UPC with a *Nvidia Titan V* and *Nvidia Titan Xp* GPUs.
- **Julia**: The Julia programming language will be used to write the code and visualize the results.

## 2.2 Gantt chart

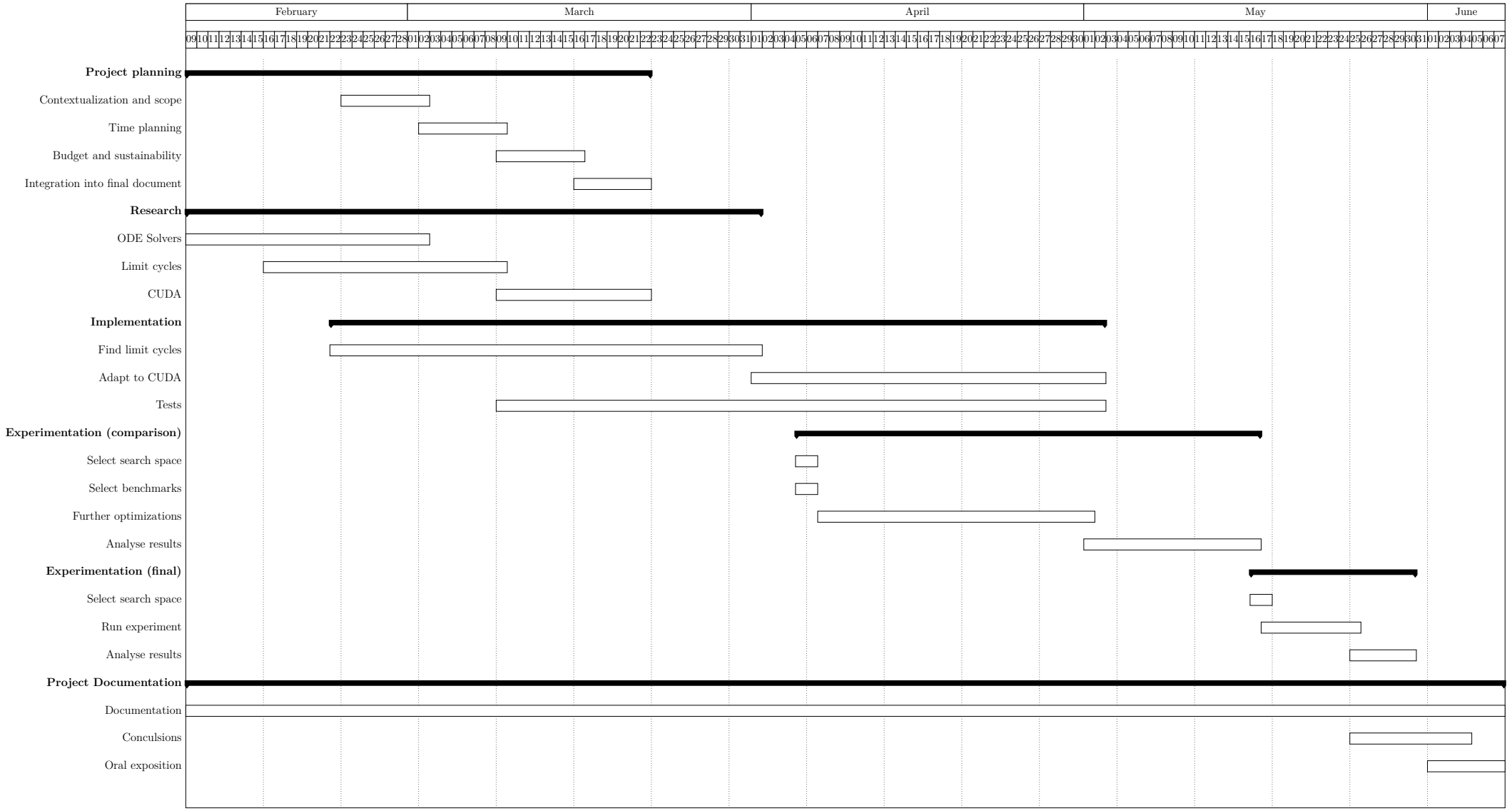


Figure 2.1: Gantt chart

## 2.3 Risk Management

As explained in section 1.4.3 there are several potential risks that may occur during the development of the project. In this section we plan how this risks will be mitigated and the steps that will be taken if they happen.

### 2.3.1 Project deadline

There is the possibility that the initial estimation made of the time it takes to complete each task was wrong due to unexpected difficulties. These delays may add up over time making it impossible to finish the project on time.

**Plan:** Detect any delays from the initial time planning immediately and remedy them by reevaluating the time planning and allocating more ours to a task. It's also crucial that there is enough time to accommodate the possible delays that may occur when planning the time for the tasks.

### 2.3.2 Computational power

**Plan:** The program will be run in a server of the department of physics of UPC. The server has 2 graphics cards that add up to 27 TFLOPS. It is important that the final version of the program is finished with enough time to be run in the server so that any potential problems with the server can be dealt with.

**Alternative:** Although it should not be a problem, in the event of not being able to use the computing power of the server, the code can be run locally on my own GPU (7 TFLOPS), it will take more time to compute but its feasible. There is also the possibility of investing on a GPU to do the calculations.

### 2.3.3 Inexperience on the field

**Plan:** Since I have limited experience with both CUDA and numerical computation of ODEs there is a significant amount of hours dedicated to studying the concepts and researching numerical methods. Devoting more hours on researching this topics will be potentially more productive than rushing into writing code without proper knowledge and will avoid mistakes in the implementation.

# 3 Budget

## 3.1 Staff costs

Although the tasks in the project are going to be performed by me and the tutors I will break them down into different roles in order to estimate the cost of the project. We will have a **Project manager** in charge of planning the project. A **Junior researcher** that will study the different computational methods and techniques needed to perform the calculations in the project. The **Junior developer** will implement the code according to the instructions given by the researcher and this code will be tested by a **Tester** to ensure that the implementation is correct. Finally, a **Technical writer** will compile all the results obtained and write the final document.

Using data from https://www.payscale.com we can estimate the cost of the different roles that take part in the project as shown in table 3.1. With these estimations we can calculate the overall cost of the staff in the project which is shown in table 3.2.

Table 3.1: Cost per hour of different roles

| **Role** | **Cost (€/h)** |
|---|---|
| Project manager | 23 |
| Junior developer | 15 |
| Junior researcher | 22 |
| Tester | 8 |
| Technical writer | 13 |

Table 3.2: Total cost of tasks

| Task | Time (h) | Role[4] | Cost (€) |
|---|---|---|---|
| **Project planning** | **60** | PM | **1,380** |
| **Research** | **150** | JR | **3,300** |
| **Implementation** | **200** | JD,T | **2,650** |
| - Different methods to find limit cycles | 100 | JD | 1500 |
| - Adapt to CUDA | 50 | JD | 750 |
| - Tests | 50 | T | 400 |
| **Experimentation (comparison)** | **130** | JR,JD | **2,160** |
| - Select search space | 5 | JR | 110 |
| - Select benchmarks | 5 | JR | 110 |
| - Further optimize the methods | 100 | JD | 1500 |
| - Analyze the results and decide the best method | 20 | JR | 440 |
| **Experimentation (final)** | **25** | JR | **550** |
| **Conclusions** | **10** | W | **130** |
| **Documentation** | **60** | W | **780** |
| **Oral exposition** | **10** | W | **130** |
| **Total** | | | **11,080** |

[4]PM = Project Manager, JR = Junior Researcher, JD = Junior Developer, W = Technical Writer

## 3.2 Generic costs

### 3.2.1 Amortization of the resources

I will work approximately 3.7 hours per day during 130 days. Most of the time on my Lenovo laptop (80%) and the rest on an HP laptop which is more lightweight and can be carried around easily. Using the formula to compute the amortization (eq. (3)) we obtain the table 3.3 which shows the amortization of the hardware used.

$$\text{Amortization} = \text{Price} \times \frac{1}{\text{Years of use}} \times \frac{1}{\text{Days of work}} \times \frac{1}{\text{Hours per day}} \times \text{hours used} \tag{3}$$

Table 3.3: Amortization of hardware

| **Hardware** | **Cost (€)** | **Life expectancy (years)** | **Amortization (€)** |
|---|---|---|---|
| Lenovo laptop | 1,400 | 6 | 211.89 |
| HP laptop | 600 | 4 | 28.38 |
| **Total** | **2,000** | | **240.27** |

### 3.2.2 Indirect costs

Apart from the hardware costs of the laptops there are more indirect costs that must be considered:

- **Internet:** With an internet cost of 100€ per month during 5 months working 3.7 hours per day the total is: $100€ \cdot 5 \cdot {}^{3.7}/_{24} = 77.08€$.
- **Electricity** Given a cost of 0.1270 €/kWh and the fact that my Lenovo laptop consumes 230 W at peak performance (which should rarely happen) we can estimate the electricity cost as: $0.1270\,€/\text{kWh} \cdot 0.230\,\text{kW} \cdot 0.5 \cdot 630\,\text{h} = 9.21€$.

In total there are 86.29€ of indirect costs which, added to the hardware amortization, result in 326.56€ of generic costs.

## 3.3 Contingency

During the development of the project unforeseen events may occur that may impact our budget. Therefore, a 15% increase on the total cost will be added as a contingency margin. Given that the Cost per Action ($CPA$) is 11,080€ and the Generic Costs ($GC$) are 326.56€ for a total of 11,406.56€. The contingency budget is then 1710.99€.

## 3.4 Incidental costs

The following table estimates the cost of the different incidents that may impact the project given their estimated cost and the risk that they happen.

Table 3.4: Incidental costs

| **Incident** | **Estimated Cost (€)** | **Risk (%)** | **Cost (€)** |
|---|---|---|---|
| Project deadline | 500 | 25 | 125 |
| Computational power | 100 | 50 | 50 |
| Inexperience on the field | 300 | 25 | 75 |
| **Total** | **900** | | **250** |

## 3.5 Final budget

Table 3.5: Final budget

| **Activity** | **Cost (€)** |
|---|---|
| $CPA$ | 11,080.00 |
| $GC$ | 326.56 |
| Contingency | 1,710.98 |
| Incidental cost | 250.00 |
| **Total** | **13,367.54** |

## 3.6 Management control

In the previous section there is an estimation of the budget and its potential risks and incidents along with their impact. We also need a model to detect the deviation on these initial budget estimations. Upon finishing a task $t$, we will calculate its deviation:

$$d_t = E_t - R_t \tag{4}$$

Where:

- $E_t$ = **Estimated cost** of the task on the initial budget plan
- $R_t$ = **Real cost** of the task when finished. Here we have to recalculate the $CPA$, $GC$, Contingency and Incidents for the task.
- $d_t$ = **Deviation from initial cost:** this estimates how much we have deviated from the original budget for each task.

If $d_t$ is negative I will reallocate the extra money for future incidents, if it's positive some funds for contingency will be used to cover the costs. On the other hand, if the contingency funds are not enough the whole budget, planning will be readdressed.

To keep track of these deviances from the budget a budget spreadsheet will be used. This spreadsheet will contain all the information on the tasks, their estimated cost, the real cost and the deviation calculations. All this budget data will be kept up to date and updated every time a task is finished. When the data is updated the deviation will immediately be apparent and the aforementioned steps will be taken.

# 4 Initial Sustainability report

## 4.1 Self-assessment

Before starting the sustainability assessment I did not know how many indicators and factors must be taken into account when doing a sustainable project. I had not considered the impact that project a part from the immediate actors involved in it. Through this assessment I realized how the project can impact the environment, the society and the economy in ways I had not thought about before.

## 4.2 Environmental impact

**Regarding PPP, Have you estimated the environmental impact of undertaking the project? Have you considered how to minimise the impact, for example by reusing resources?**

The main impact of this project on the environment is the use of computer resources and therefore electric power to do the complex calculations needed. The aim of the project is not only to develop a program that works but that it also uses the least amount of resources so that it does not waste computing power. In order to avoid wasting resources, only the final version will be run on a big search space and during development the tests will be on a much reduced search space.

**Regarding the life expectancy, How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution environmentally improve existing solutions?**

Currently, there is not much work on the research of the number of limit cycles in second degree ODEs and the current approaches rely on heady computations in order to calculate the cycles. With this new approach the aim is to search with a faster implementation although less accurate which will hopefully find interesting systems that can then be analyzed in more detail.

## 4.3 Economy

**Regarding PPP, Have you estimated the cost of undertaking the project (human and material resources)?**

In section 3 there is a description of the cost of the project taking into account human and material resources as well as potential risks and contingencies.

**Regarding the life expectancy, How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution economically improve existing solutions?**

As stated on the Sustainability section, if a faster more efficient method to identify limit cycles is implemented it will reduce the amount of computational power needed and therefore the resources.

## 4.4 Social

**Regarding PPP, What do you think undertaking the project has contributed to you personally?**

The project enables me to research on various topics that interest me and has given me a new view on the applications of GPUs for scientific research.

**Regarding the life expectancy, How is the problem that you wish to address resolved currently (state of the art)? In what ways will your solution socially improve (quality of life) existing Is there a real need for the project?**

The nature of this project as a research project on the field of dynamical systems does not have a direct impact on the society as a whole. However, if interesting results are obtained it could bring some insight into Hilbert 16th problem which has remained unsolved for more than a century.

# 5 Program structure

Initially, all the program was to be implemented in pure *Julia* since there are libraries to interface with *CUDA* [20] that should be able to handle the tasks needed. However, there were some complications with the *Nvidia* drivers and compiling the *Julia CUDA* kernels and some features are not yet available in *CUDA.jl*.

Therefore, the main program in *CUDA* will be implemented in *C* and expose the relevant method through an *ABI* (Application Binary Interface) so that it can be compiled into a shared binary and interface with *Julia* or any other language capable of using *C* shared libraries. With this approach we can achieve seamless interoperation between *CUDA* and *Julia* while having full control of how *CUDA* kernels behave and manage the resources. All the analysis on the data obtained from the *CUDA* computation can then be processed using all the available *Julia* libraries offering great flexibility.

The *ABI* interface will be very simple, exposing a method to initialize the GPU memory, and the various different kernels to compute the data and transfer the result to *Julia*. Since *Julia* is able to use *C* structures natively through its *C* interface there is no overhead when transferring data since pointers can be shared [6] and all the memory management can be delegated to *C*.

# 6 Initial implementation

Before implementing the program in CUDA, we first need to implement a sequential version of the program. This sequential code allows easier debugging of the implementation and provides a basis to compare the speedup obtained with CUDA.

The main steps that the program must perform are:

1. Compute a trajectory starting at various different points.
2. Determine if the trajectory is a limit cycle.
3. Classify all the starting points into groups corresponding to the different limit cycles.

The first two steps of computing the trajectory and determining if it is a limit cycle will be parallelized using *CUDA* and the last step of analysis will be done using *Julia*.

## 6.1 Computing the trajectory

To compute the trajectory, a numerical method to integrate an ODE must be used. There are various methods that can be used with varying complexity. In the program we implemented *Runge-Kutta* methods of different order [1, 2, (2)3, 4 and 4(5)] [5]. Two of these methods allow for adaptive time stepping [(2)3 and 4(5)] meaning that they adjust the time steps ($h$) according to the error tolerance.

There is no need to save the complete trajectory in order to detect the presence of the limit cycles since the relevant metrics can be computed at the same time as the integration is being performed. This reduces not only the amount of memory needed to compute and store the results but also allows the early termination of a trajectory computation if we detect that it is a cycle, point or it goes out of bounds before the last step is reached.

In section 8 there is an in-depth discussion of the metrics used and how they are employed to detect limit cycle candidates. For the moment the relevant point is that for each trajectory we only need to compute 2 *loops* and measure the ratio between the maximum and minimum points of each loop. If this ratio is equal to 1 (within a tolerance margin), the trajectory is a cycle.

### Julia

For the very first prototyping, the various methods where implemented in *Julia* to check the correctness of the algorithm and then manually translated into *C* and checked again.

### C translation

In listings 1 and 2 we provide examples of how the main RK4 routine is implemented in *Julia* and what is its corresponding *C* translation. Both versions of all methods were tested to ensure that they where properly implemented (In section 7 there is a detail comparison of the accuracy of the methods).

```
function RK4(f::Function, x, y, h)
    xk1, yk1 = h.*f(x, y)
    xk2, yk2 = h.*f(x + xk1/2, y + yk1/2)
    xk3, yk3 = h.*f(x + xk2/2, y + yk2/2)
    xk4, yk4 = h.*f(x + xk3, y + yk3)

    x = x + (xk1 + 2xk2 + 2xk3 + xk4)/6
    y = y + (yk1 + 2yk2 + 2yk3 + yk4)/6

    x, y
end
```

Listing 1: Julia version of RK4 step

```
__device__ void rk4_step(const double x0, const double y0, double *const x1, double
↪ *const y1, const double h, const double P[PARAMS]) {
    double cache[4][2];

    // xk1, yk1 = h.*f(x, y)
    f(x0, y0, h, &cache[0][0], &cache[0][1], P);

    // xk2, yk2 = h.*f(x + xk1/2, y + yk1/2)
    *x1 = x0 + cache[0][0]/2.0;
    *y1 = y0 + cache[0][1]/2.0;
    f(*x1, *y1, h, &cache[1][0], &cache[1][1], P);

    // xk3, yk3 = h.*f(x + xk2/2, y + yk2/2)
    *x1 = x0 + cache[1][0]/2.0;
    *y1 = y0 + cache[1][1]/2.0;
    f(*x1, *y1, h, &cache[2][0], &cache[2][1], P);

    // xk4, yk4 = h.*f(x + xk3, y + yk3)
    *x1 = x0 + cache[2][0];
    *y1 = y0 + cache[2][1];
    f(*x1, *y1, h, &cache[3][0], &cache[3][1], P);

    // x = x + (xk1 + 2xk2 + 2xk3 + xk4)/6
    // y = y + (yk1 + 2yk2 + 2yk3 + yk4)/6
    *x1 = x0 + (cache[0][0] + 2*cache[1][0] + 2.0*cache[2][0] + cache[3][0])/6.0;
    *y1 = y0 + (cache[0][1] + 2*cache[1][1] + 2.0*cache[2][1] + cache[3][1])/6.0;
}
```

Listing 2: C version of RK4

# 7 Convergence analysis

Once the sequential code is implemented, we must analyze the results obtained to verify the correctness of the program. Therefore, a convergence analysis was performed in which we compared the results obtained with our implementation to the theoretical results of a well known ODE.

$$\begin{aligned} \frac{dx}{dt} &= -y \\ \frac{dy}{dt} &= x \end{aligned} \tag{5}$$

We used the test system eq. (5) for which all trajectories are known to have a circular shape with center $(0, 0)$ and a period of $T = 2\pi$. For each method we found the numerical trajectory $\mathbf{r}(t)$ which depends on the specific method used and the timestep. The error is calculated as the absolute value of the displacement after one period $T$,

$$\varepsilon = |\mathbf{r}(T) - \mathbf{r}(0)| \tag{6}$$

In fig. 7.1 we show how the error depends on the timestep $\Delta t$ and on the specific integration scheme. This allows to verify the consistency in the order of the integration scheme. For order $p$ of accuracy, the error scales as

$$\varepsilon = \mathcal{O}((\Delta t)^p) \tag{7}$$

As expected, the higher-order methods have higher order of accuracy. Additionally, for the higher order methods there is a point in which the numerical error of the method becomes irrelevant since it is less than the *machine epsilon*. The specific value of the machine epsilon depends on the type of variable, CPU architecture and compiler. In our case using the precision of `double` we find that for the Runge-Kutta of order 4 the smallest useful timestep is $\Delta t \approx \frac{2\pi}{10^4}$. A further decrease in the timestep actually worsens the accuracy as this implies a summation of a larger number of values, thus increasing the importance of the rounding error.

The error (eq. (6)) was calculated by taking into account the exact value of the period $T$. In a more general situation, the period might not be known. Thus we perform a different study in which the error is estimated as the difference in the maximal $x^{\max}$ position between two consequent periods, $x_i^{\max}$ and $x_{i+1}^{\max}$

$$\varepsilon = |x_{i+1}^{\max} - x_i^{\max}| \tag{8}$$

We computed the error using values of $\Delta t$ non commensurate with the period of the circle and interpolated the period using a second degree polynomial. The results are shown in fig. 7.2 and are quite different to fig. 7.1 since Euler and Midpoint methods appear to have order 2 and RK3, RK4, RKF45 order 3. This is probably due to the polynomial interpolation used and other interpolation methods should be considered to obtain better results.

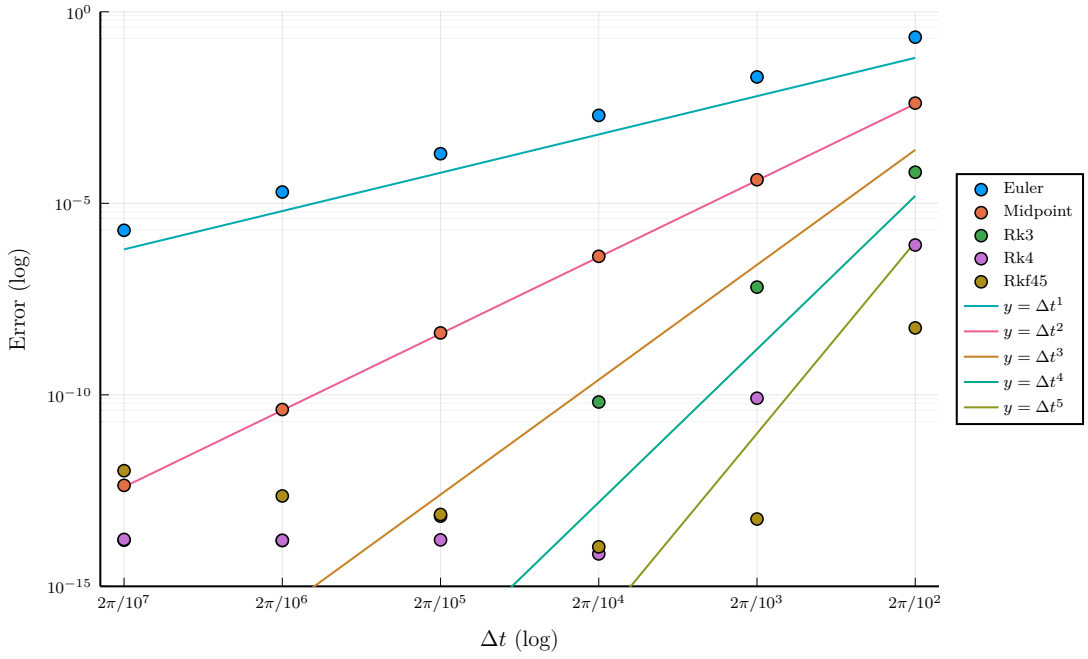


Figure 7.1: Integration error after one period with commensurate $\Delta t$, estimated according to eq. (6).

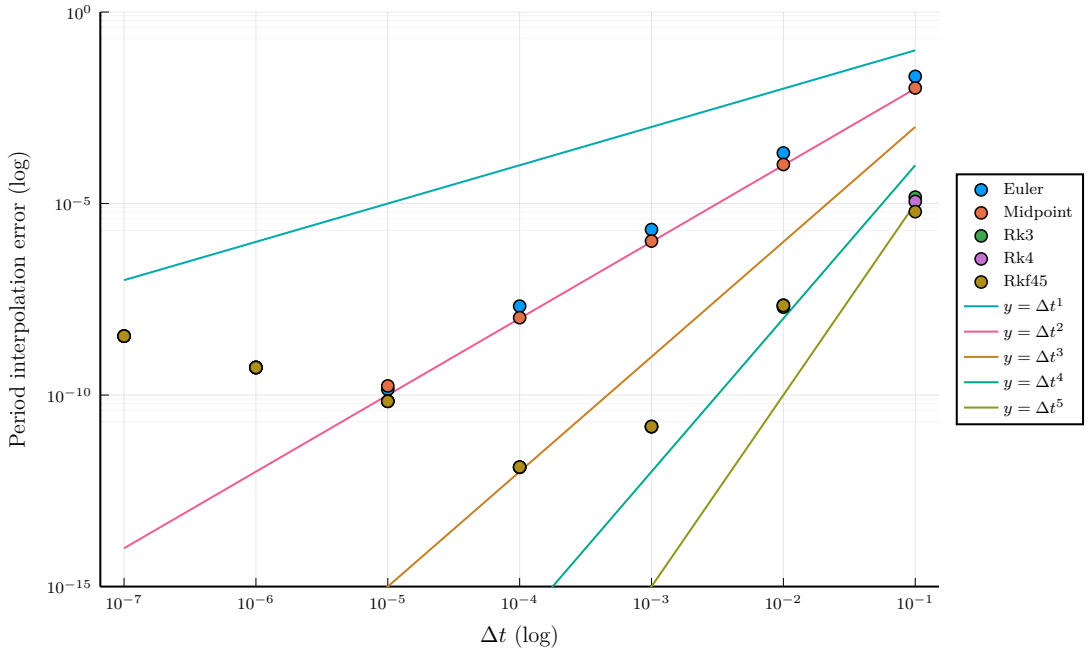


Figure 7.2: Error on the period estimation using interpolation, estimated according to eq. (8)

# 8 Metrics

As discussed in previous sections, there is no need to save the full trajectory in order to detect limit cycle candidates; we can evaluate the trajectory's characteristics in-place.

To do so we have to detect special points in the trajectory that we can use to detect if we are in a cycle. These points must be easy to identify (computationally cheap) and should be present in all trajectories. By comparing the values of these points at each loop of the cycle we can estimate the behaviour of the trajectory.

**Zero crossings:**

can be computed by detecting the change of sign during the computation. Although they are really cheap and are commonly used on complex ODE systems, there is no guarantee that cycles will cross axis.

**Inflection points:**

these correspond to the points where the ratio of change of the function changes sign (maxima, minima or stationary points). Given that the integration methods used compute the result as the previous value plus a change, we can easily detect changes in sign with almost no overhead. Moreover, all cycles will have at least four local extrema.

**Interpolation:**

although we can detect if we passed an inflection point with almost no overhead, we must perform some kind of interpolation to obtain an accurate value within our tolerances. To do so, we can perform interpolation using 3 neighbouring points or perform additional integration steps to find the inflection point numerically. Applying polynomial interpolation requires saving at least 3 point for each trajectory, interpolating them and computing the vertex. The second option (root finding) is potentially more costly since it involves additional integration steps but should produce better results.

## 8.1 Rate of change of extrema

Comparing the value of the initial 4 local extrema with the local extrema of the second loop of the cycle we can obtain a rate of change of the trajectory. For instance, given the values of the 4 local extrema on obtained on the first loop: $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ and their counterparts obtained on the second loop: $x'_{\min}, x'_{\max}, y'_{\min}, y'_{\max}$ we can compute the ratio between each of them to obtain their rate of change. If the ratio is less than 1, the trajectory is *decreasing*, if it is greater than 1 it is *increasing* and if it is 1 (within an adequate tolerance) the trajectory is a cycle.

There is one small caveat with this approach which is that the ratios between the 4 different extrema are not comparable, that is, in some cases the maxima or minima are various orders of magnitude apart. This has to be taken into consideration when evaluating the values used for the tolerance when searching the points with ratio 1.

Traditionally with CPU computation one initial point is taken and evaluated through many steps until the trajectory reaches a stable cycle. Instead, the advantage of GPU computation is that one can take a massive amount of points, evaluate only the very few first steps (until 2 loops are completed) and quickly find limit cycle candidates.

## 8.2 Other metrics

Apart from the rate of change in local extrema, an evaluation of the rate of change in the loop period was also attempted, but it did not give satisfying results.

## 8.3 Distinguishing different cycles

The rate of change of extrema allows us to find areas in the function with trajectories with a rate of change of their extrema of 1 (section 8.1), which are candidates to be limit cycles. Now the problem is how we can classify these areas into different limit cycle trajectories. The following approaches were tested:

### Connected component labelling

Connected component labelling is a technique often used in the field of computer vision. It consists of finding connected components in a graph and labeling each pixel according to the component in which they belong [37].

Since the final result of our computation is a matrix with ones and zeros indicating the coordinates close to trajectories with rate of change 1, we have a binary image in which we can apply this technique. Ideally the connected components found will match the limit cycles, but there are two main caveats: there may be areas where resolution is not high enough (the discretization of the section was too big to and the points sampled missed the trajectory) and the line can be partially broken, thus splitting a cycle into two or more parts. The second problem appears when two cycles pass really close to each other and be classified as the same group. Therefore, this technique requires very small discretization of the area studied which requires computing lots of points.

### Clustering

Given that we are computing the ratio of four local extrema points, we can use these same extrema values to analyse the characteristics of the trajectory. To do so, a part from returning the ratio of the extrema points, the four extrema points themselves will also be returned. With this information we have four points defining a cycle which we can classify using any clustering technique. Three different clustering methods where tested:

### Hierarchical clustering

Using hierarchical clustering we obtained correct clusters in the reference system of [22]. However, the this algorithm does not scale well when large number of points is used. It has a runtime complexity of ($\mathcal{O}(n^3)$) where $n$ is the number of points analyzed, this makes it impractical for moderate to larger cases.

### Kmeans clustering

As with hierarchical clustering, with *Kmeans* we can use the four values of local extrema when computing the clustering. The algorithm minimizes the square

distance of the points in the clusters with their centers as shown in eq. (9) where $k$ is the number of clusters, $n$ the number of points, $x^{(j)}$ is a point belonging to cluster $j$, $c_j$ is the center of cluster $j$ and $\|\|$ the normal of a vector.

$$J = \sum_{j=1}^{k} \sum_{j=1}^{n} \|x_i^{(j)} - c_j\|^2 \tag{9}$$

**??? what is J? add it to the text**

The runtime complexity of *Kmeans* method of $n$ points with $k$ clusters and $d$ dimensions is $\mathcal{O}(knd)$ as shown in ref. [2]. It is exponential in the worst case, but since we know that if there are limit cycles the algorithm should converge quickly (the clusters will be clear **??? rephrase** ), we can limit the number of iterations to small values.

With *Kmeans* we must specify the number of clusters the search, it does not say how many clusters there are in the data. To do so we must compute the algorithm with 2, 3, 4 and 5 clusters and use a metric that indicates which partition is better.

We will use the silhouette metric, defined in eq. (10). Here $s(i)$ is the silhouette coefficient of the point $p_i$. It is calculated from the average distance $a(i)$ from the point $p_i$ to all the points in the cluster to which $p_i$ belongs and $b(i)$ which is the minimum average distance to all the clusters in which $p_i$ does not belong. If the silhouette value is $s(i) = 1$, it indicates that the point $p_i$ is close to all the points in his cluster and far away from the others. The worst value is $s(i) = -1$. We compare the mean of the silhouette coefficient for the different number of clusters and find which gives the best result. **check the paragraph, I edited it ???**

$$\begin{aligned} a(i) &= \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i,j) \\ b(i) &= \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i,j) \\ s(i) &= \begin{cases} 1 - a(i)/b(i), & \text{if } a(i) < b(i) \\ 0, & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1, & \text{if } a(i) > b(i) \end{cases} \end{aligned} \tag{10}$$

# 9 CUDA implementation

The C implementation was adapted to be run by CUDA. Each CUDA thread computes a full trajectory to completion and threads run in parallel as long as we do not exceed the GPU threads. For the initial version we implemented the program to run in a single GPU with a total of 10240 CUDA cores and 12Gb of memory [5]

Given the results of the convergence analysis from section 7, each trajectory was computed using RK4 with a step of $\Delta t = 10^{-3}$ performing $10^4$ steps. To save the full trajectory we need $10^4 \cdot 8$ bytes (80kb). To compute 10240 trajectories the total amount of memory needed is around 800Mb which is feasible. However, we can compute the different metrics to find limit cycles in place during the integration steps, so there is no need to save the full trajectory (we can just save 8 to 100 bytes of data for each trajectory depending on the metrics we compute).

Given a pair of axis (delimited $x$ and $y$ ranges) the different points in the grid were separated into blocks of 32 by 32 points (1024 threads) and ran using a CUDA kernel that computed the different values wanted for each trajectory. For simplicity, the divisions performed on each axis where equal for $x$ and $y$.

Although we have no problems on the amount of memory needed to save the information needed, there are limitations in the number of register variables available to each block (group of threads). If too many variables are needed to perform the computation, they can cause a runtime error and the variables will need to be moved to the heap (*spilled*) or the threads will have to be reduced. If using The maximum number of registers per block, the threads have a maximum of 75 registers. The most complex kernel implemented used 54 registers but when combined with rk45 stepping there were variables *spilled* into the shared memory. For this reason, the *Runge Kutta* of oder 3 without adaptive stepping was used by default. The results where similar with a reduced execution time and less memory usage.

[5] Details on the GPU hardware can be found in listing 3 in the appendix

# 10 Results

## 10.1 Replicating results from Kutznetsov et al.

To analyze the performance of the program and verify it correctness the parameters found by Kutznetsov et al. [22] are used. [6]

Specifically, the following system of equations is studied,

$$\begin{aligned} \frac{dx}{dt} &= x^2 + xy + y^2 + x + y \\ \frac{dy}{dt} &= ax^2 + bxy * cy^2 + \alpha x + \beta y \end{aligned} \qquad \begin{aligned} a &= -10 \\ b &= 2.2 \\ c &= 0.7 \\ \alpha &= -72.7778 \\ \beta &= 0.0015 \end{aligned} \tag{11}$$

This specific choice of the coefficients is known to result in four limit cycles, as shown in fig. 10.1.

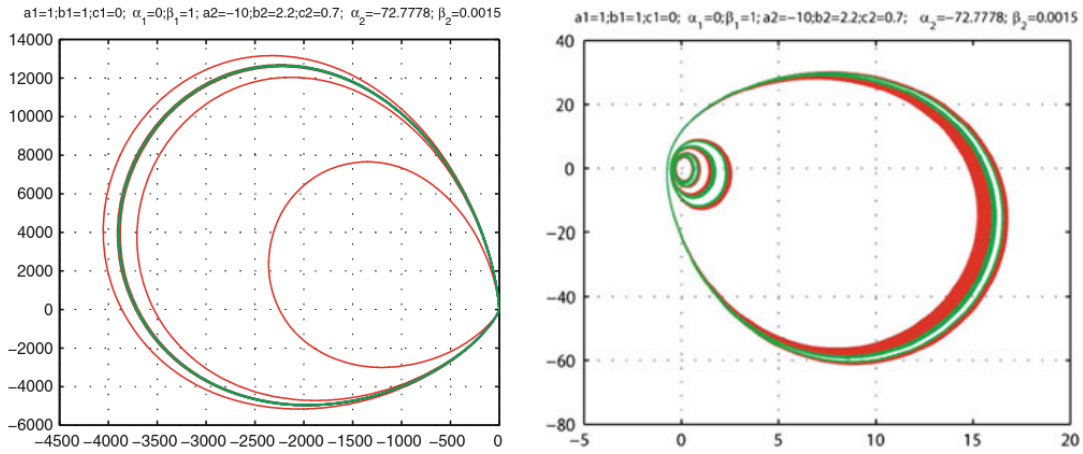


Figure 10.1: Visualization of four limit cycles in two-dimensional polynomial quadratic system, from Ref. [22]

[6] This system is the same that was previously discussed in section 1 (fig. 1.1)

Figure 10.2 shows a visualization of the points we obtain with a rate of change of extrema equal to 1 ($\pm 10^{-6}$). The coefficients are chosen to be equal to these of Ref. [22] and their results are reproduced. The computation of this data took 250ms on a grid of $1024 \times 1024$ (i.e. approximately one million of trajectories) using *Runge Kutta* of third order with a step size of $10^{-3}$. Notice that some points from the left part ($x < 0$) of the biggest cycle are not found.

Nevertheless, three limit cycles are clearly identified. Notice that the 4th limit cycle which is expected to be present in the left of our plot cannot be found by the program due to the stiffness of the system on that area. This problem is discussed in more detail in section 12. For the rest of the analysis we will only consider these 3 limit cycles.

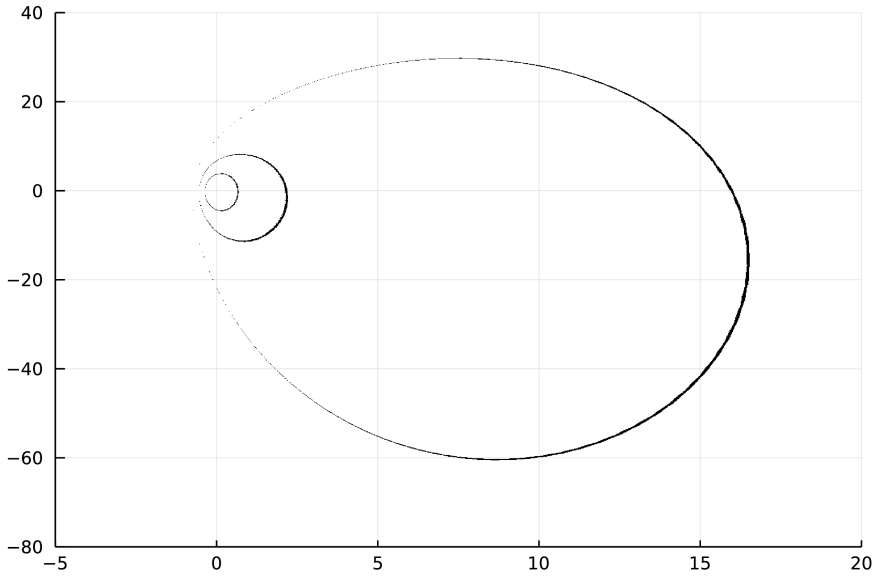


Figure 10.2: Limit cycles found in the reference case, Eq. 11, discretizing the phase space by $1024 \times 1024$ grid.

Using a bigger grid size ($5000 \times 5000$ initial points), we obtain clearer results and the execution time grows to 1200ms=1.2s. The increase of the resolution does not lead to drastic change in the picture as the limit cycles were clearly separated on a grid with a smaller resolution. In fig. 10.4 we can appreciate how having 5 times more resolution on each dimension gives more detail which could be crucial in some other cases in which the limit cycles are not so clearly visible with the specified window. In section 10.3 there are some examples of some possible situations where resolution is crucial to obtain good results.

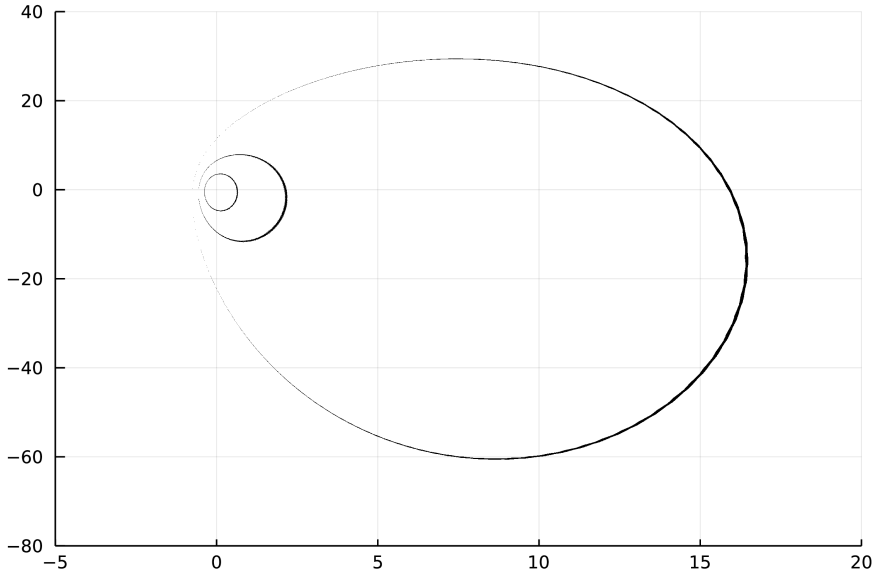


Figure 10.3: Limit cycles found in the reference case, Eq. 11, discretizing the phase space by $5000 \times 5000$ grid.

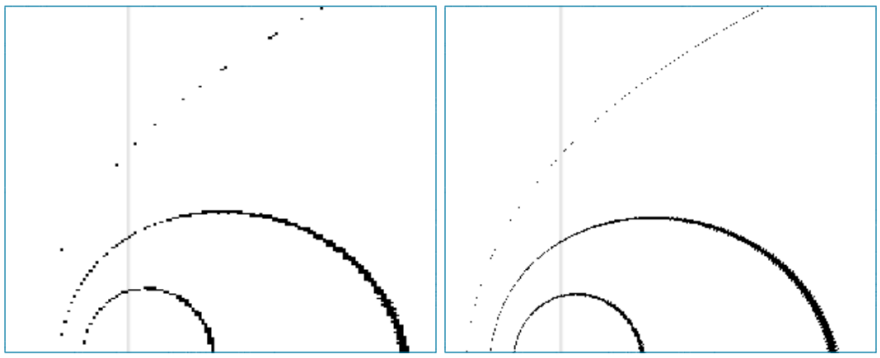

Figure 10.4: Detail of the same region from fig. 10.2 (left) and fig. 10.3 (right)

### Clustering

Figure 10.5 shows the histogram of the positions of the four local extrema, that is the maximal and minimal positions of the $x$ and $y$ coordinates. One can clearly see that there 3 groups in each one of the local extrema associated with 3 different

limit cycles. Thus this type of the histogram allows to count the number of the limit cycles if they are present.

If we compute the *K-means* for 3 clusters we obtain the cluster centers shown in table 10.1. Figure 10.6 shows the bounding boxes defined by the clusters obtained as an overlay over fig. 10.2.

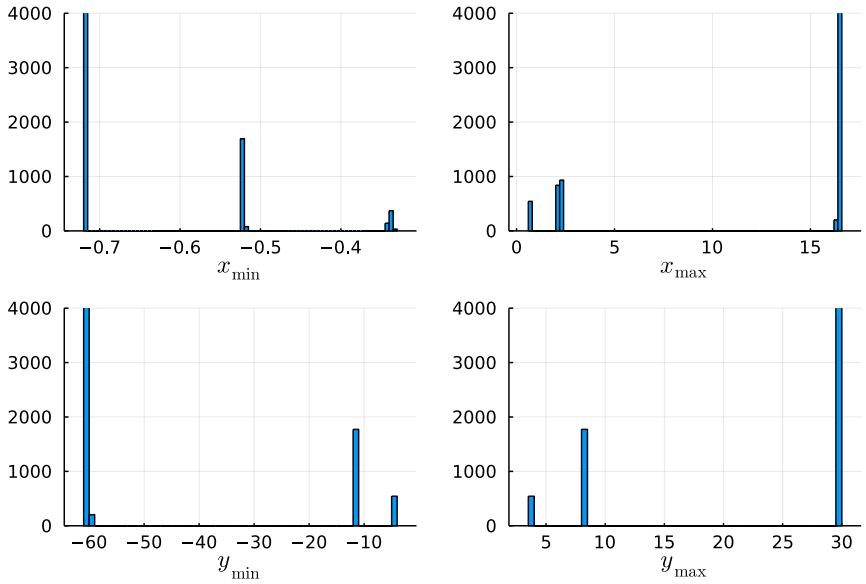


Figure 10.5: Histogram of distribution of values for each local extrema with rate of change equal to 1 (tol $1^{-6}$).

Table 10.1: Cluster centers

| Cluster | $x_{\min}$ | $x_{\max}$ | $y_{\min}$ | $y_{\max}$ |
|---|---|---|---|---|
| A | -0.716937 | 16.4835 | -60.2806 | 29.7273 |
| B | -0.521985 | 2.20031 | -11.3237 | 8.17958 |
| C | -0.338311 | 0.68522 | -4.48625 | 3.84119 |

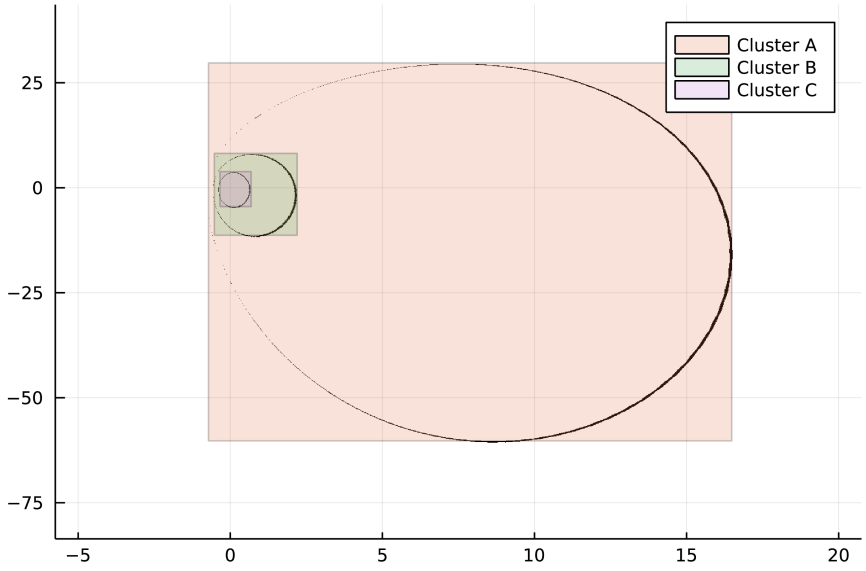


Figure 10.6: Phase portrait. Bounding boxes defined by the clusters in table 10.1 overlaid on top of fig. 10.2

## 10.2 Scanning the values of the coefficients

Figure 10.7 shows the phase portrait used for finding limit cycles in the same system used in the previous examples (eq. (11)) but with a slight modification of parameter $a$ from 10 to 10.1. We can see that the cycles are slightly more similar in size. Changing to 10.2 only one cycle remains. With $a = 9.9$ there are only two cycles. Other slight modifications of the other parameters also reduce the number of cycles.

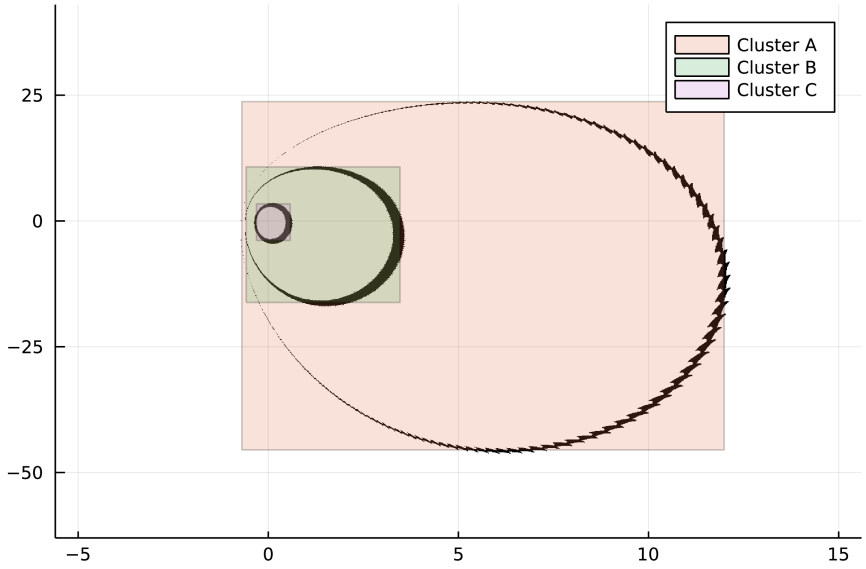


Figure 10.7: Phase portrait of the system for parameters as in eq. (11) but with parameter $a$ modified from $10.0 \rightarrow 10.1$. Three limit cycles are visible.

## 10.3 Changing the window range

One of the important issue is a possible dependence of the number of the limit cycles on the size of the area which is used for searching them. The previous figures were obtained by searching for limit cycles inside a rectangle bound by $x \in (-5, 20), y \in (-60, 40)$. In the following section an analysis on the capacity of the program to find the cycles with different windows will be performed.

### Bigger area

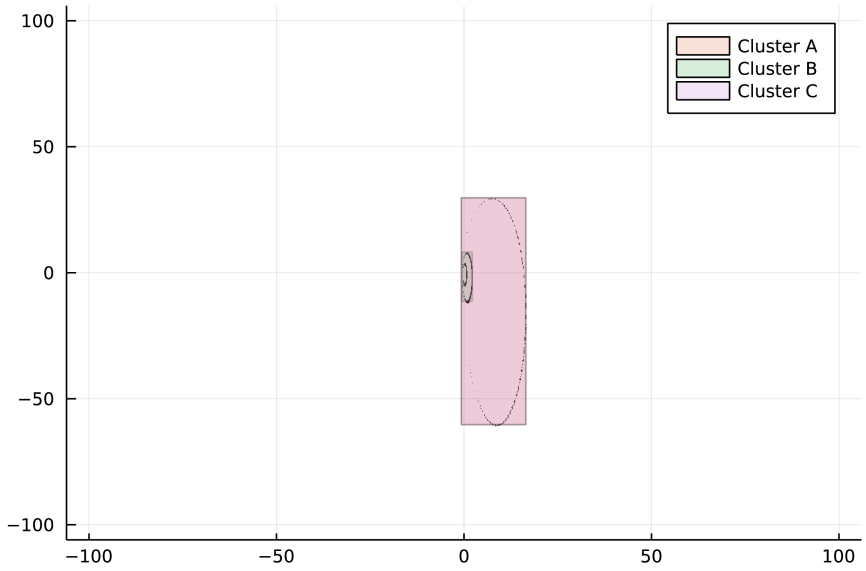


Figure 10.8: Limit cycles found on system from eq. (11)
($1024 \times 1024$ grid over $x, y \in (-100, 100)$)

As shown in fig. 10.8, with a window of $x, y \in (-100, 100)$ we obtain the clusters without problem. Increasing the window further to $(-1000, 1000)$ while maintaining the same grid size does only find the two bigger limit cycles. A possible optimization could be to make two steps: one to find the biggest limit cycles and a second one using as a window the bounding box of the limit found.

### Partial occlusion

A good algorithm should be robust to partial occlusion of the cycle, meaning that even if part of the cycle is outside the window it should still be detected. In fig. 10.9 we show an image of the results applied with a window of $x \in (0, 5), y \in (0, 20)$ (Shown in gray). Indeed, the clusters are equivalent to the ones in the previous examples. Note that the line of the biggest cluster is very thin since there are not many points. As long as the window contains enough part of the trajectory of a limit cycle[7] with enough resolution, the cycles can be identified.

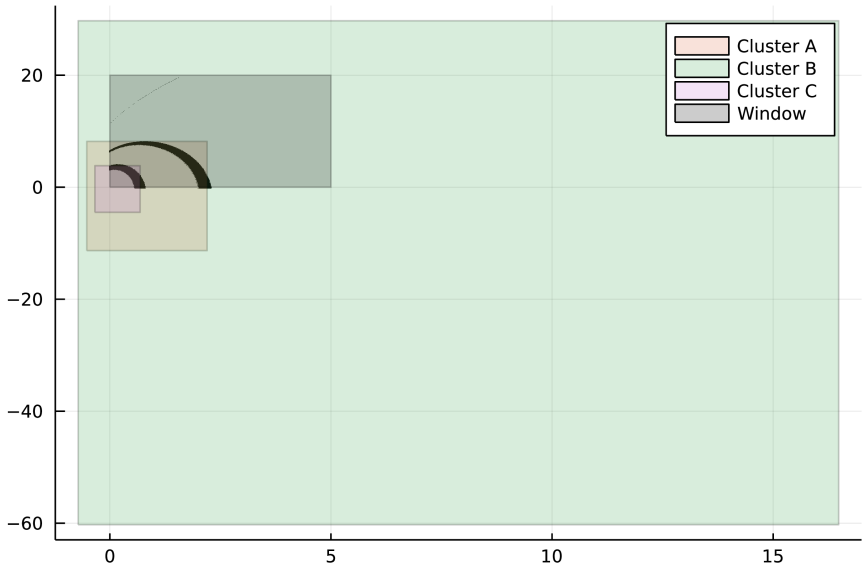


Figure 10.9: Limit cycles found on system from eq. (11)
($1024 \times 1024$ grid over $x \in (0, 5), y \in (0, 20)$)

[7] And the trajectory is not stiff (see section 12)

### Compactified plane

Since the $(x, y)$ plane in $\mathbb{R}^2$ is infinite it is not possible to cover it with a finite regular grid. A possible way out is to do a compactification so that the infinitely large plane is mapped to a finite-size box. In section 12 there is a more in depth explanation of how this can be done. A simple way to perform the compactificaction $(x, y) \to (x', y')$ is to use mapping $x' = \arctan x$, $y' = \arctan y$ where $x \in (-\infty, \infty)$ to $x' \in (-\pi/2, \pi/2)$. This compactification is not ideal as it significantly deforms the plane and does not preserve the angles, but still it allows to perform the search of cycles in essentially infinite phase space.

However, this compactification does not help if they are big since they get *squished* on the borders meaning that without a very high resolution it is not possible to find them. For instance, if we divide the range $(-\pi/2, \pi/2)$ in 4096 parts, in the outermost point before $+\infty$: correspond to $\tan(\pi/2 - \pi/4096) \approx 1304$ which is approximately 1304, the immediately previous value is $\tan(\pi/2 - 2\pi/4096) \approx 652$. If our limit cycle does pass between these values it will not be found.

The result of applying the tangent compactification and running the program is shown in fig. 10.10. The biggest cycle is almost non visible despite the fact that it is not particularly big ($y_{\min} \approx -60, y_{\max} \approx 30$).

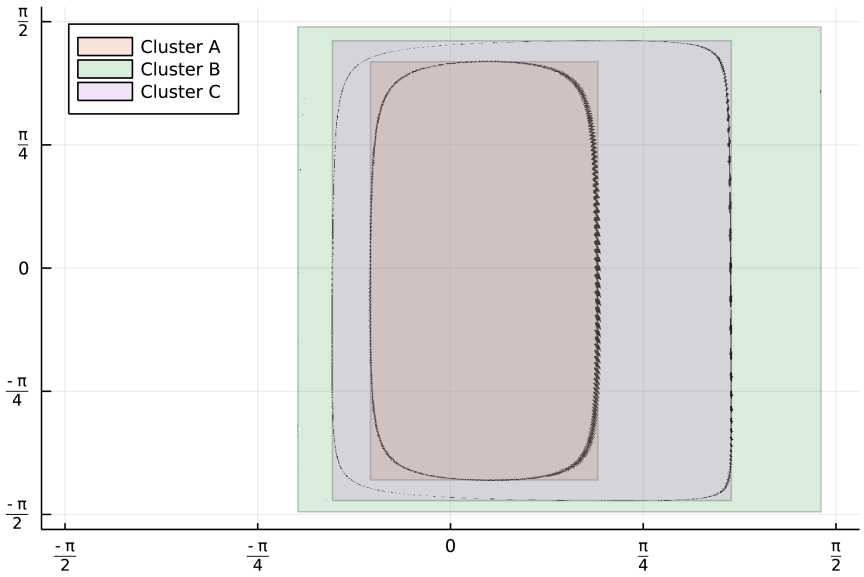


Figure 10.10: Limit cycles found on system from eq. (11)
($1024 \times 1024$ grid over compactified plane)

# 11 Performance analysis

In this section we analyze the runtime performance of the CUDA part of the program compared to the same implementation sequentially in C. All these measures are done using the same kernel that computes the ratio of local extrema for each point. The grid size refers to the number of points in the grid: a grid of size $2^14$ corresponds to a grid with $2^7$ points in both $x$ and $y$ ($2^7 \times 2^7$). For each of these $2^14$ points a trajectory is computed. In these benchmarks the trajectories were performed using the *Runge Kutta* of order 3 with a step size of 0.0001 and 10000 steps.

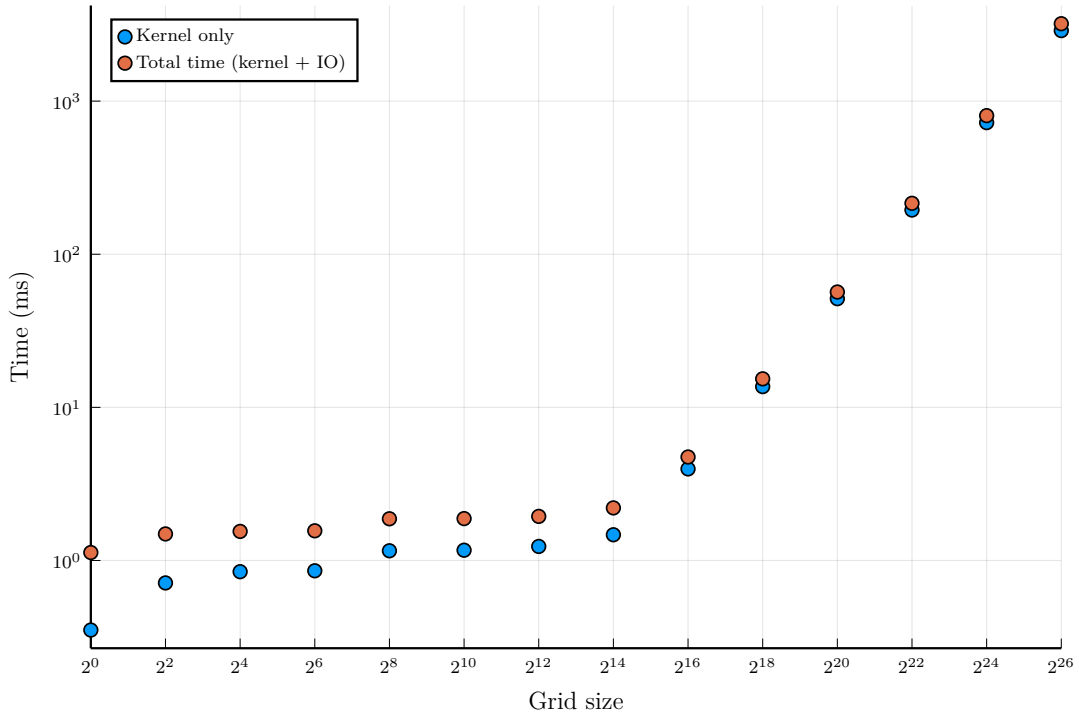


Figure 11.1: Execution time of CUDA version with separate kernel time

Figure 11.1 shows the execution time of the CUDA version of the code with separate measurements of the time spent in the kernel (i.e. useful time of the actual calculation). Total time is the time spent in the kernel plus the additional time for input/output operations and CUDA synchronization directives. For grid sizes of 1 and 2, most of the time is spent outside the kernel. We can observe that the time is roughly the same from grid sizes 1 to $2^14(16384)$ but after that there is a

notable increase. This is due to the fact the GPU used has 10240 CUDA cores, so we are using all cores and reaching the parallelization limit of the device.

If we now add the execution time of the *C* sequential implementation of the program we obtain data shown in fig. 11.2. The CUDA version outperforms the sequential version for grid resolutions as low as $4 \times 4$. This is due to the nature of the computation which has very little overhead on copying operations between the GPU and CPU, allowing for really notable performance gains. The quadratic fit on the sequential time shows how it increases quadratically with the grid resolution.

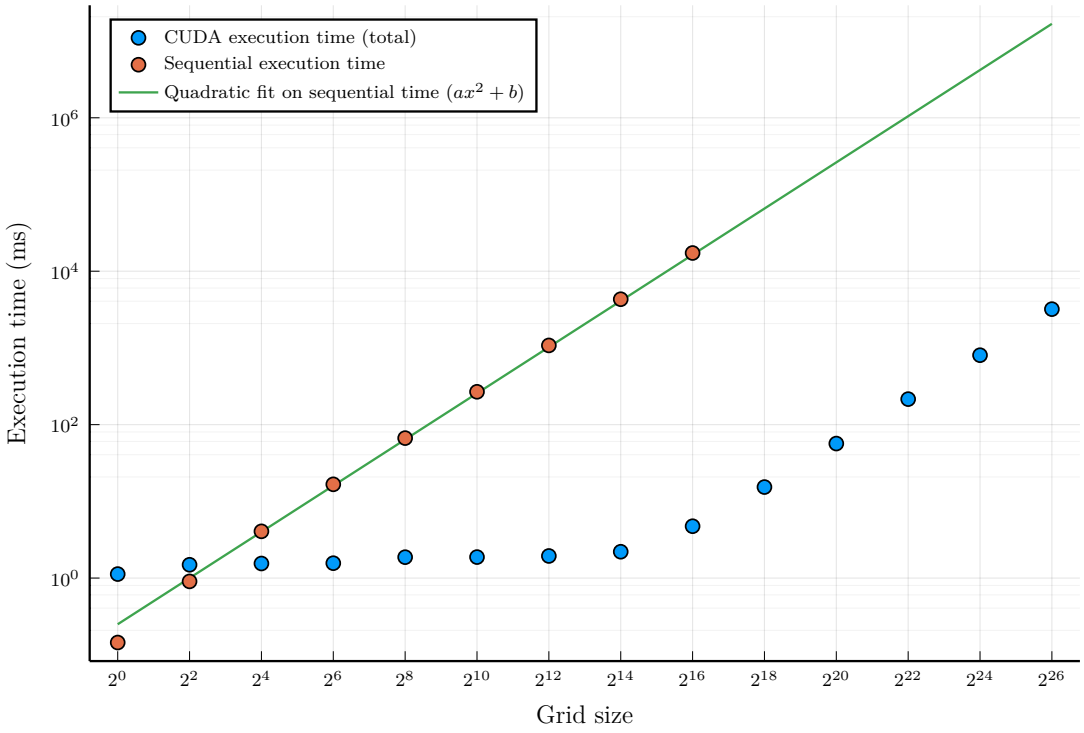


Figure 11.2: Execution time of CUDA version vs. Sequential

### Speedup

By computing the ratio between the execution time of the sequential version and the CUDA version, we obtain the *Speedup* (defined in eq. (12)), which indicates how much faster the CUDA computation is compared to the sequential version. As stated before the GPU used has 10240 cores so the theoretical maximum speedup is 10240. Obviously such speedup is not feasible, both due to the fact that a CPU

core runs faster than a CUDA core and that there are additional operations of communication between CPU and GPU that increase the execution time. The dependence of the speedup on the number of trajectories is shown in fig. 11.3.

$$S = \frac{T_{\text{original}}}{T_{\text{improved}}} \tag{12}$$

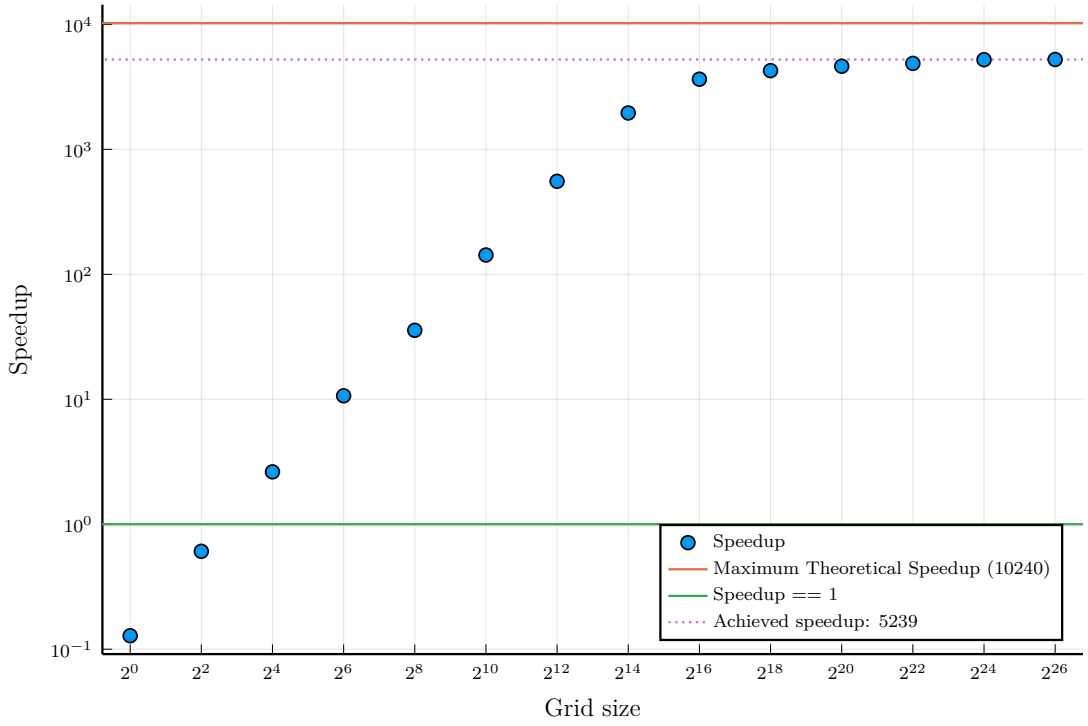


Figure 11.3: Speedup of CUDA version vs. Sequential

The overhead for starting CUDA processes make the GPU code slower for calculation of small number of trajectories. For $\approx 2^4 = 16$ trajectories the parallel and the serial codes require a similar execution time. The parallel code runs faster (speedup larger than one) for larger number of trajectories. The theoretical maximum for speedup is limited by the number of cores in GPU (10240 in the present case) while here we observe 5239 at maximum. There is still a factor of 2 to reach the maximum achievable speedup which gives room for improvement. With additional GPUs the factor will be different. For reference, the computation of $2^{26} \approx 6 \times 10^7$ trajectories took 3.202s on the GPU while the sequential version of the program would take approximately 4 hours and 40 minutes. Such a massive

computation of $2^{26}$ trajectories requires around 7 Gb of GPU memory if using and polynomial interpolation (although there is only 550Mb of data to needed for the final result). Given that the GPU has 12 Gb of memory, higher resolutions cannot be performed with this kernel.

## 11.1 Multi GPU

As discussed in the previous section, the limitation on memory on the GPU makes it impossible to compute easily the phase portrait with higher resolutions. One solution is to employ various GPUs for the task. To demonstrate the capability, the code was adapted dividing the area to compute into sectors of equal size and computing each part on one GPU, then merging the results. The server used in the other experiments only has 2 GPUs and the second one has much lower capabilities than the first so the benefits of using the 2 GPUs were outweighed by the overhead of passing data to the GPUs and the fact that the grid had to be split unevenly giving most of the work to the faster GPU. Thanks to the department of computer architecture of the FIB, the code was run in their *boada* server with 4 GPUs. These GPUs only have 2880 CUDA cores (as opposed to the 10240 of the Titan V) [8] therefore, with the combined power of 4 GPUs the theoretical maximum is 11520 which is the same order of magnitude as the Titan V.

Figure 11.4 shows the speedup obtained in *boada* using 1 and 4 GPUs. We can observe that even with 4 GPUs they still don't match the speedup obtained with the Titan V. The overhead of using multiple GPUs is quite significant, so there is no benefit until grid sizes $2^{20}$. The speedup of using 4 GPUs in the case of $2^{26}$ is $2981/750 \approx 3$. Using that as a reference using 4 Titan GPUs we could expect a speedup of $\approx 15700$. This was just done as a proof of concept that it is technically possible.

[8] See listing 4 for the details on *boada* hardware

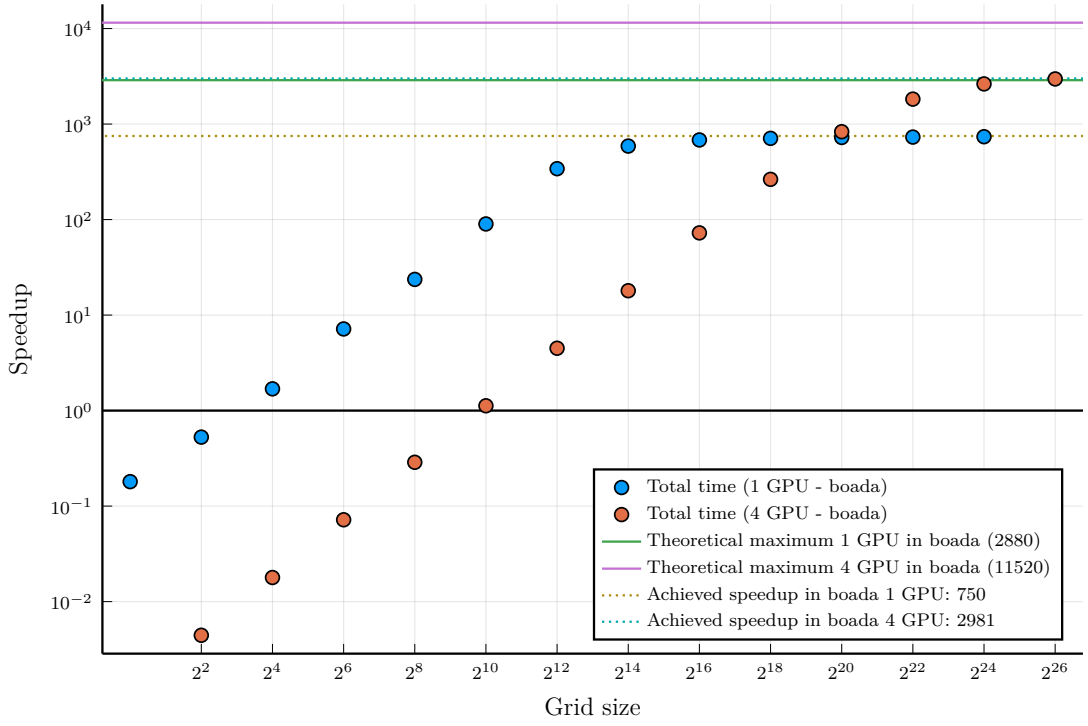


Figure 11.4: Speedup of CUDA version vs. Sequential

# 12 Compactification and Stiffness

One problem that arises when trying to find limit cycles is where to look for them in the space. Since the $\mathbb{R}^2$ plane is infinite, we must decide what boundaries of the plane to evaluate. There are at least two methods to solve this issue: compactification (compacting the infinite field into a finite one through mathematical constructs) and analytical analysis of singular points interest.

### Compactification

There are various methods of compactification that can be used to study limit cycles, one of these is the Poincaré, Bendixon or Poincaré Lyupanov compactification [30, 3, 11, 12] among others. These methods reduce the infinite field into a semi-sphere that that can be projected onto a plane for easier visualization and analysis. For simplicity, we compactified the plane using the tangent function: a point $x$ in the infinite space can be mapped to $\alpha \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$ such that: $x = \tan(\alpha)$. This produces great distortion on the plane making all trajectories seem squares and values of $x$ greater than 1000 are indistinguishable.

### Singular points

Performing analytical analysis to find points of interest is quite complex and requires vast knowledge on the field of ODEs and limit cycles. Moreover, to perform analytical computation special software to handle symbolic maths is required (like Maple [25]). Ideally these points could be computed beforehand using normal CPU computation and use them to decide the areas that need analysis using our program. However, it is known that all limit cycles in a quadratic system contain a node [8]. Indeed, in the system we are analysing $(0, 0)$ will always be a node, therefore limit cycles should appear around the origin. There may be more nodes depending on the parameters of the system, but we know that the area around the origin is a good candidate.

### Stiffness

Another problem which arises when numerically integrating ODE systems is the so called *stiff* system where traditional *Runge-Kutta* integration methods struggle due to the numerical instability of the system. In particular, the higher the order of the *Runge Kutta* method, the less stability it has [38]. The only solution to be able to integrate stiff systems (or stiff sections of a system) is to have specialized methods for stiff integration, detect when a trajectory is stiff and use these methods instead [36].

There are various stiff integration methods, but their implementation in *CUDA* is non-trivial since they all involve some sort of equation solving at each step [5].

# 13 Interactive application

As discussed in section 5, the core of the program is written as a *C* library which exposes functions through *ABI* (Advanced Binary Interface), allowing other languages capable of interpreting *C* structures to interact with it natively. This combined with the really fast computation times makes it really easy to integrate the functionality of the program into an interactive application to research how limit cycles behave in a system.

## 13.1 Pluto notebooks

Pluto notebooks are *Julia* web notebooks designed around the concept of *javascript* observables, they build a dependency tree between all cells and if a cell is modified the change propagates to the rest [29]. This allows to develop reactive notebooks. Figure 13.1 shows an example of a *Pluto* notebook that uses the program to calculate limit cycles with given parameters and displays the result. Any change in the parameters is propagated and the plot is updated seamlessly in around 1 second [9].

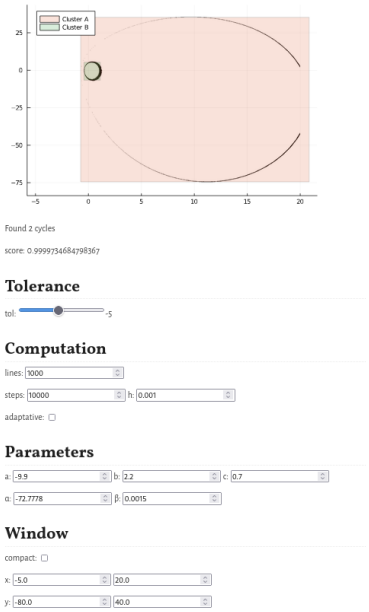


Figure 13.1: Example of interactive Pluto notebook showing 2 limit cycles

[9] With reasonable parameters

## 13.2 OpenGL

Another potentially useful feature is the interoperability between *CUDA* and *OpenGL*, allowing to bind the actual data used in *CUDA* in the GPU to *OpenGL* textures which can be rendered directly to the screen [27]. Using little modification to the program an interactive render of the GPU data can be done.


Figure 13.2: OpenGL window directly displaying CUDA results

## 13.3 CUDA limitations on interactivity

One of the main problems with the current implementation is that changing the system involves creating a new *CUDA* function and recompiling the library, this makes using the program as a library quite challenging since the use must know how to program the function and compile it. Without using *CUDA* we could simply use function pointers so that any arbitrary function can be passed and no need for recompilation is needed. However, function pointers are extremely slow in *CUDA* since the function cannot be *inlined* and we introduce massive overhead by adding function calls to the stack.

One option is to implement a program to parse ODE systems with a strict syntax, produces the *CUDA* kernel compiles it and dynamically links it to the program. It should be possible to do from *Julia*.

# 14 Parameter search

A parameter search was performed on the system, to obtain a table that related the different parameter combinations with the number of limit cycles found. Given that there where cases where a limit was not detected, the hope was

There where 5 parameters so the search space is quite large even if we only consider a few possible values. Determining the number of cycles of a system took $\approx 1.5$ seconds using a $2048 \times 2048$ grid (depending on the characteristics of the system).

## 14.1 Compactified

Since there was not enough time to compute a wide search we only applied variations from the original parameters in eq. (11) one at a time. 2500 different values where tried for each parameter (keeping the others as normal). This gives a total of $2500 * 5$ different systems to try which at 1.5 seconds took around 5 hours and a half. We used tangent compactification a step size of $1^{-3}$, 20000 steps, tolerance $10^{-6}$ and *Runge Kutta* of order 3.

## 14.2 Not compactified

We also performed a smaller parameter search without applying compactification using a window of $x, y \in (-50, 50)$ with the same search configuration.

## 14.3 Results

Table 14.1 shows the parameter search results for the two searches performed. Unfortunately non of the searches found 4 or more cycles, which is to be expected since we only varied one parameter at a time so the system was always quite close to the original. However it shows that the compactification does obtain more results than using an arbitrary window.

Table 14.1: Parameter search results

| | 2 cycles | 3 cycles | $\geq 4$ cycles | Total analyzed |
|---|---|---|---|---|
| Compactified | 648 | 172 | 0 | 12500 |
| (-50, 50) | 158 | 119 | 0 | 12500 |

Despite not finding 4 or more limit cycles some of the results where sampled from the ones obtained to check if indeed they contained the limit cycles specified. In most cases the number was correct. There were a few exceptions caused by applying the *Kmeans* on too little points. However these can easily be filtered out by comparing the differences between the sizes of the cycles found.

Figure 14.1 shows one of the results that gave limit cycles with close proportions. This was found in the parameter search using the compactified plane. In this system all the cycles are closer in size. Note that the changes in the parameter are minimal.

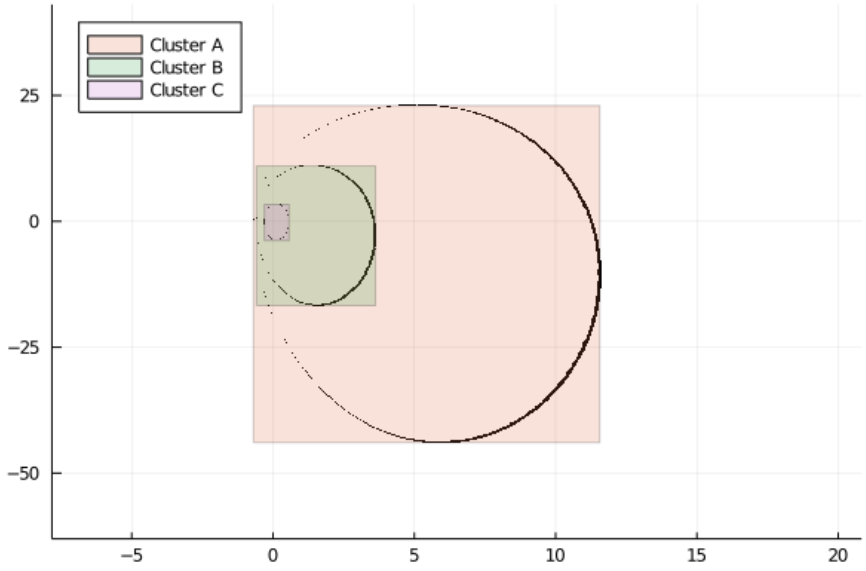


Figure 14.1: Limit cycles found on system from eq. (11)
With modified parameter $c = -72.05$

In the search all the systems of 3 cycles found where with very similar values to the ones in eq. (11). The systems with 2 cycles where much more diverse. In fig. 14.2 there is one system in which with $a = -27.7$ there are 2 limit cycles found, one very small around the origin is and a much bigger one to the left. This left limit cycle is the one which with higher parameters of $a$ we cannot detect because it increases exponentially and the trajectories become unstable. However finding this left cycle shows that with a capable integrator which can handle stiff trajectories the cycle should be found in other systems.

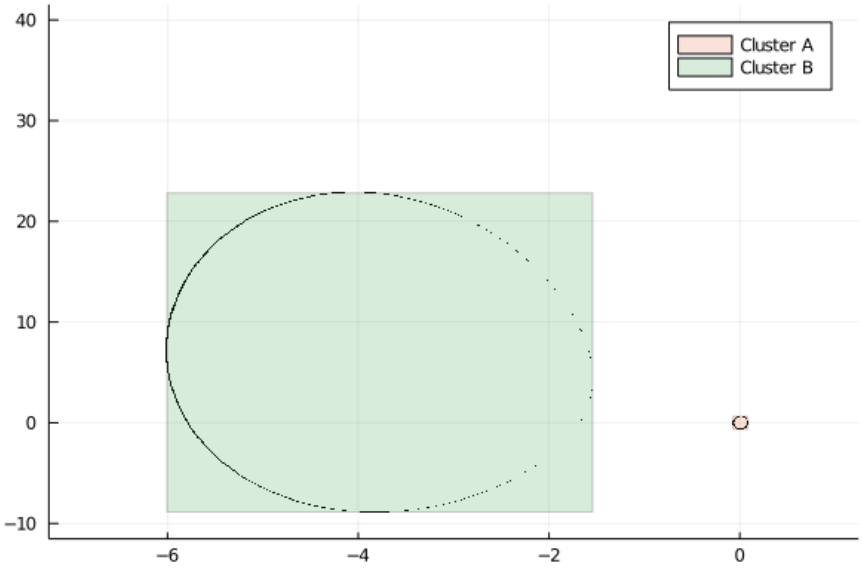


Figure 14.2: Limit cycles found on system from eq. (11)
With modified parameter $a = -27.7$

# 15 Conclusions

With this project we were able to develop and implement a fast method to assist in the detection of limit cycles by leveraging the computational power of modern GPUs. The program is much faster than finding limit cycles by integrating a trajectory until it converges, which can take a lot of time depending on the initial conditions and the system.

Initially the program was implemented in the *Julia* programming language, but the *CUDA* part of the program was finally implemented in *C* do to the limited documentation and capabilities of *Julia*'s *CUDA* API. Despite being written in *C*, its was integrated into *Julia* through an *ABI* which allows the user to use the advanced abstraction and libraries of *Julia* to visualize and further process the data obtained.

The program can also be used with a powerful GPU to perform a parameter search to find interesting systems. The parameter search performed has been tiny compared to the initial objective due to the limited resources and time available but can be easily scaled to use multiple GPUs as shown in section 11.1. The nature of the algorithm and its simplicity allows to scale it indefinitely.

Unfortunately, not all limit cycles on a system are found reliably, there are a couple improvements that could be made in the future to fix this issue. In section 16 there is a discussion on the future of the project.

All in all, the project is an initial proof of concept on how the processing power of a GPU can be used in the study of limit cycles. This opens the door for further research of the topic using more advanced methods.

# 16 Further work

Despite obtaining promising results, many improvements can be made. In particular, possible analytical methods that can help narrow down locations of possible limit cycles might be developed. Another area of the project that could be improved is the integrator methods for stiff equations. There are hundreds of well studied and documented integration methods that far outperform the simple *Runge Kutta* methods used in the project. Porting the implicit integrators from GNU Scientific Library to *CUDA* as some researchers have done with other parts of the library [33] could enable much better integration for stiff systems.

The *UAB* (Universitat Autonoma de Barcelona) has an open source program named *P4* [34, 35] to analyze ODE systems numerically and analytically. It has the ability to find limit cycles by computing trajectories between two points given by the user until it reaches a cycle. This method is quite slow, in fact in ref. [10] there is an explicit mention on how you should specify points very close by and reduce the precision of the computation since otherwise the limit cycle detection may take a lot of time. The following quote from page 267 illustrates the issue:

> A *Searching for limit cycles* window appears with a time bar which should show the time left for computing but whose most useful application is to stop searching, since it may easily delay a lot before or after finding a limit cycle.

Maybe our approach could be added as an alternative to assist in finding limit cycles. Combining our program with the various analytical techniques of *P4* may open new possibilities for GPU computation of other constructs a part from limit cycles.

Other possible improvements could be the analysis of systems of higher degree and dimensionality which are much more difficult to visualize and investigate analytically. For example, there have been found systems of degree 3 with 11 limit cycles [15].

# 17 Final sustainability report

## 17.1 Environmental Impact

### PPP (project put into production)

The environmental impact can be quantified in the number of hours of usage of the server used to run the program. According to the server logs, there have been around 170 hours of server usage from the account I used. It is difficult to estimate accurately how this translates to energy usage, but most of the time was spent developing the program and running tests on a single system. The two parameter searches performed where also quite small and ran for around 5 and 12 hours each using around 200 Watts on the GPU. All these estimates do not take into account the computing time spent on my own machine or the multi-GPU execution on the *boada* server, but overall the environmental impact was quite small.

If I were to carry the project out again, I would probably be able to use less computing resources in the initial stages of the development of the program. During the first iterations of the development cycle, I ran various tests that didn't really work and took quite some time to compute.

### Exploitation

The main resources needed to use the project are the GPUs needed. As of now, there are two possible use cases for the program: helping find limit cycles of a unique system, or performing a parameter search to find systems with interesting configurations of limit cycles. The former can done with a simple *CUDA* capable GPU and requires *minimal* resources. The latter is much more costly depending on the number of parameters to search, it could require a powerful GPU or even a bunch of GPUs.

The program makes good use of the GPU resources, as such, it should be more energy efficient than running the same method on a CPU or the method of computing a single trajectory from a point until convergence used to find limit cycles with a CPU.

#### Risks

The program is unable to find limit cycles in some cases, so it may have to be improved in the future. If the proposed new integrators to be able to find those cycles are implemented, there could be a change in the resources needed to run the program. Nonetheless, it will probably have minimal impact, since now there is still a need to run inefficient CPU code to search stiff areas of a system.

## 17.2 Economic Impact

#### PPP

Overall, the initial budget outlined in section 3 was a good estimation of the cost of the project, there were no major problems that required modifications to the budget nor was there any need for the contingency budget.

#### Exploitation

As we pointed out in the *PPP* section, the main cost of the project is the computational resources to run it (what GPUs and how many are needed) as well as their power consumption.

Right now the project needs some polishing to reach its full potential, therefore an update may be needed in the future, probably to address the improvements discussed in section 16. This will require human resources to work on the program and maintain it.

#### Risks

Currently, the program has been tested with a very few systems. A rigorous test on the numerical stability of the integrators used was performed, but that is not enough to guarantee there may not be issues with some systems.

## 17.3 Social Impact

### PPP

Undertaking this project I have learned a lot about general purpose graphics processing unit programming (GPGPU) and scientific computing.

### Exploitation

As we mentioned on section 16 there are researchers using similar tools to explore constructs on ODEs and quick identification of limit cycles could help speedup their research.

The program developed in the project does identify limit cycles quickly, but has trouble when dealing with some trajectories. It can assist on finding limit cycles, but it is not a complete replacement to other methods. Some parameter searches were performed but all of them only managed to find 3 limit cycles.

### Risks

This project uses the proprietary *CUDA* platform. Therefore, it can only be executed on *Nvidia* GPU. This makes the users dependent on this vendor since they cannot run the program on other GPUs.

# References


[1] Antaviana, Can. *NVIDIA CUDA Technology Dramatically Advances The Pace of Scientific Research - News*. IMIM Institut Hospital del Mar d'Investigacions Mèdiques. Dec. 2008. URL: https://www.imim.cat/news/37 (visited on 03/01/2021).

[2] Arthur, David, Manthey, Bodo, and Röglin, Heiko. "k-Means has Polynomial Smoothed Complexity." In: *arXiv:0904.1113 [cs]* (Aug. 7, 2009). arXiv: 0904.1113. URL: http://arxiv.org/abs/0904.1113 (visited on 06/19/2021).

[3] Bendixson, Ivar. "Sur les courbes définies par des équations différentielles." In: *Acta Mathematica* 24 (none Jan. 1901). Publisher: Institut Mittag-Leffler, pp. 1–88. ISSN: 0001-5962, 1871-2509. DOI: 10.1007/BF02403068. URL: https://projecteuclid.org/journals/acta-mathematica/volume-24/issue-none/Sur-les-courbes-d%c3%a9finies-par-des-%c3%a9quations-diff%c3%a9rentielles/10.1007/BF02403068.full (visited on 06/17/2021).

[4] Besard, Tim, Foket, Christophe, and De Sutter, Bjorn. "Effective Extensible Programming: Unleashing Julia on GPUs." In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (Apr. 1, 2019), pp. 827–841. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112.

[5] Butcher, John C. *Numerical Methods for Ordinary Differential Equations*. New York, 2008. ISBN: 978-0-470-72335-7. URL: https://handwiki.org/wiki/Runge%E2%80%93Kutta_methods.

[6] *C Interface · The Julia Language*. URL: https://docs.julialang.org/en/v1/base/c/ (visited on 06/17/2021).

[7] Casades, Gemma and De la Llave, Rafael. "Computation of Limit Cycles and Their Isochrons: Fast Algorithms and Their Convergence." In: *SIAM Journal on Applied Dynamical Systems [electronic only]* 12.4 (Jan. 1, 2013), pp. 1763–1802. DOI: 10.1137/120901210.

[8] Cherkas, Leonid A, Artes, Joan C, and Llibre, Jaume. "Quadratic systems with limit cycles of normal size." In: (2003), p. 16.

[9] "Hilbert's 16th Problem and Its Weak Form." In: *Limit Cycles of Differential Equations*. Ed. by Christopher, Colin and Li, Chengzhi. Advanced Courses in Mathematics CRM Barcelona. Basel: Birkhäuser, 2007, pp. 95–109. ISBN: 978-3-7643-8410-4. DOI: 10.1007/978-3-7643-8410-4_11.

[10] "Examples for Running P4." In: *Qualitative Theory of Planar Differential Systems*. Ed. by Dumortier, Freddy, Llibre, Jaume, and Artés, Joan C. Berlin, Heidelberg: Springer, 2006, p. 276. ISBN: 978-3-540-32902-2. DOI: 10.1007/978-3-540-32902-2_10. URL: https://doi.org/10.1007/978-3-540-32902-2_10 (visited on 06/17/2021).

[11] "Poincaré and Poincaré–Lyapunov Compactification." In: *Qualitative Theory of Planar Differential Systems*. Ed. by Dumortier, Freddy, Llibre, Jaume, and Artés, Joan C. Berlin, Heidelberg: Springer, 2006, pp. 149–163. ISBN: 978-3-540-32902-2. DOI: 10.1007/978-3-540-32902-2_5. URL: https://doi.org/10.1007/978-3-540-32902-2_5 (visited on 06/17/2021).

[12] *Fig. 1. Poincaré compactification*. ResearchGate. URL: https://www.researchgate.net/figure/Poincare-compactification_fig1_226646278 (visited on 06/09/2021).

[13] Gasull, Armengol, Giacomini, Héctor, and Grau, Maite. "Effective localization of limit cycles (from numerical to analytical results)." First joint meeting Brazil-Spain, Fortaleza, 2015. URL: http://www.gsd.uab.es/1jmbsm2015/Gasull.pdf.

[14] GitHub. *Managing your work on GitHub*. GitHub Docs. 2021. URL: https://docs.github.com/en/github/managing-your-work-on-github (visited on 03/22/2021).

[15] Han, Maoan and Li, Jibin. "Lower bounds for the Hilbert number of polynomial systems." In: *Journal of Differential Equations* 252.4 (Feb. 15, 2012), pp. 3278–3304. ISSN: 0022-0396. DOI: 10.1016/j.jde.2011.11.024. URL: https://www.sciencedirect.com/science/article/pii/S0022039611004943 (visited on 06/21/2021).

[16] Hilbert, D. "Mathematical problems." In: *Bulletin of the American Mathematical Society* 8.10 (July 1, 1902), pp. 437–480. ISSN: 0002-9904. DOI: 10.1090/S0002-9904-1902-00923-3.

[17] Hilbert, D. "Mathematische Probleme." In: *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse* 1900 (1900), pp. 253–297. URL: http://eudml.org/doc/58479.

[18] Hoff, Quay van der, Greeff, Johanna C., and Kloppers, P. Hendrik. "Numerical investigation into the existence of limit cycles in two-dimensional predator-prey systems." In: *South African Journal of Science* 109.5 (Jan. 2013). Publisher: Academy of Science of South Africa, pp. 01–06. ISSN: 0038-2353. URL: http://www.scielo.org.za/scielo.php?script=sci_arttext&pid=S0038-23532013000300013 (visited on 03/01/2021).

[19] Ilyashenko, Yulij. "Centennial History of Hilbert's 16th Problem." In: *Bulletin of The American Mathematical Society* 39.3 (July 1, 2002), pp. 301–355. DOI: 10.1090/S0273-0979-02-00946-1.

[20] *JuliaGPU*. URL: https://juliagpu.org/ (visited on 06/17/2021).

[21] Klöckner, Andreas et al. "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation." In: *Parallel Computing* 38.3 (Mar. 2012), pp. 157–174. ISSN: 01678191. DOI: 10.1016/j.parco.2011.09.001.

[22] Kuznetsov, N. V., Kuznetsova, O. A., and Leonov, G. A. "Visualization of Four Normal Size Limit Cycles in Two-Dimensional Polynomial Quadratic System." In: *Differential Equations and Dynamical Systems* 21.1 (Jan. 1, 2013), pp. 29–34. ISSN: 0974-6870. DOI: 10.1007/s12591-012-0118-6.

[23] Leonov, G. A. and Kuznetsov, N. V. "Hidden attractors in dynamical systems. From hidden oscillations in Hilbert–Kolmogorov, Aizerman, and Kalman problems to hidden chaotic attractor in Chua circuits." In: *International Journal of Bifurcation and Chaos* 23.1 (Jan. 2013), p. 1330002. ISSN: 0218-1274, 1793-6551. DOI: 10.1142/S0218127413300024.

[24] Llibre, Jaume. "Sobre el problema 16 de Hilbert." In: *La Gaceta de la Real Sociedad Matemática Española* 18.3 (2015), pp. 543–554. URL: https://gaceta.rsme.es/abrir.php?id=1289.

[25] *Maple - The Essential Tool for Mathematics - Maplesoft*. URL: https://www.maplesoft.com/products/maple/ (visited on 06/17/2021).

[26] NVIDIA. *CUDA Toolkit Documentation*. 2021. URL: https://docs.nvidia.com/cuda/index.html (visited on 03/22/2021).

[27] *OpenGL Interoperability*. URL: http://docs.nvidia.com/cuda/cuda-driver-api/index.html (visited on 06/19/2021).

[28] Papadimitriou, Christos H. and Vishnoi, Nisheeth K. "On the Computational Complexity of Limit Cycles in Dynamical Systems." In: *arXiv:1511.07605 [cs, math]* (Nov. 24, 2015). arXiv: 1511.07605.

[29] Plas, Fons van der. *fonsp/Pluto.jl*. original-date: 2020-02-23T01:50:12Z. June 19, 2021. URL: https://github.com/fonsp/Pluto.jl (visited on 06/19/2021).

[30] Poincaré, Henri. "Sur l'intégration algébrique des équations différentielles du premier ordre et du premier degré." In: *Rendiconti del Circolo Matematico di Palermo* 5 (1891), pp. 161–191.

[31] Rackauckas, Christopher. *A Comparison Between Differential Equation Solver Suites In MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran*. type: dataset. 2017. DOI: 10.15200/winn.153459.98975.

[32] Rackauckas, Christopher and Nie, Qing. "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia." In: *Journal of Open Research Software* 5 (May 25, 2017), p. 15. ISSN: 2049-9647. DOI: 10.5334/jors.151.

[33] rodrigo. *GNU Scientific Library (GSL) partially ported to NVIDIA GPUs: cuSL*. Black Hole Group. Feb. 7, 2019. URL: https://blackholegroup.org/2019/02/07/gnu-scientific-library-gsl-partially-ported-to-nvidia-gpus-cusl/ (visited on 06/17/2021).

[34] Saleta, Oscar. *oscarsaleta/P4*. original-date: 2016-11-15T11:46:19Z. July 20, 2018. URL: https://github.com/oscarsaleta/P4 (visited on 06/17/2021).

[35] Saleta, Oscar et al. *Computer program P4 to study Phase Portraits of Planar Polynomial differential equations*. 2018. URL: https://mat.uab.cat/~artes/p4/p4.htm (visited on 06/17/2021).

[36] Shampine, L. F. and Hiebert, K. L. "Detecting stiffness with the Fehlberg (4, 5) formulas." In: *Computers & Mathematics with Applications* 3.1 (Jan. 1, 1977), pp. 41–46. ISSN: 0898-1221. DOI: 10.1016/0898-1221(77)90112-2. URL: https://www.sciencedirect.com/science/article/pii/0898122177901122 (visited on 06/17/2021).

[37] Shapiro, Linda G. "Connected Component Labeling and Adjacency Graph Construction." In: *Machine Intelligence and Pattern Recognition*. Ed. by Kong, T. Yung and Rosenfeld, Azriel. Vol. 19. Topological Algorithms for Digital Image Processing. North-Holland, Jan. 1, 1996, pp. 1–30. DOI: 10.1016/S0923-0459(96)80011-5. URL: https://www.sciencedirect.com/science/article/pii/S0923045996800115 (visited on 06/19/2021).

[38] Skvortsov, Leonid. "Accuracy of Runge–Kutta Methods Applied to Stiff Problems." In: *Computational Mathematics and Mathematical Physics* 43 (Sept. 1, 2003), pp. 1320–1330.

# A Appendix

```
Device 0: "TITAN V"
  Major revision number:                         7
  Minor revision number:                         0
  Total amount of global memory:                 12652838912 bytes
  Number of multiprocessors:                     80
  CUDA Cores/MP:                                 128
  Total CUDA Cores                               10240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.46 GHz
  Memory Clock rate:                             0.85 GHz
  Memory Bus Width:                              3072 bits
  Number of asynchronous engines:                7
  It can execute multiple kernels concurrently:  Yes
  Concurrent copy and execution:                 Yes
```

Listing 3: GPU hardware details

```
Devices 0,1,2,3: "Tesla K40c"
 Major revision number:                         3
 Minor revision number:                         5
 Total amount of global memory:                 11996954624 bytes
 Number of multiprocessors:                     15
 CUDA Cores/MP:                                 192
 Total CUDA Cores                               2880
 Total amount of constant memory:               65536 bytes
 Total amount of shared memory per block:       49152 bytes
 Total number of registers available per block: 65536
 Warp size:                                     32
 Maximum number of threads per block:           1024
 Maximum sizes of each dimension of a block:    1024 x 1024 x 64
 Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
 Maximum memory pitch:                          2147483647 bytes
 Texture alignment:                             512 bytes
 Clock rate:                                    0.75 GHz
 Memory Clock rate:                             3.00 GHz
 Memory Bus Width:                              384 bits
 Number of asynchronous engines:                2
 It can execute multiple kernels concurrently:  Yes
 Concurrent copy and execution:                 Yes
```

Listing 4: boada GPU hardware details