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

Author: Francesc Forn Cañabate

Director: Grigori Astrakharchik, Marina Gonchenko

Bachelor Thesis
Specialisation in Computer Science

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Abstract

The main goal of this Bachelor's thesis is to numerically tackle the second part of the famous Hilbert's 16th problem, particularly the detection of limit cycles in two-dimensional systems of quadratic ordinary differential equations (ODEs). We developed and successfully verified a high-performance algorithm to concurrently target this problem, harvesting the computational capabilities of modern Graphical Processing Units (GPUs).

In this project we studied the performance and precision of different numerical methods for solving the system of differential equations, and have chosen the best algorithm with optimal parameters for the best convergence and performance. We developed and programmed a numerical solver based on the fourth-order Runge-Kutta integration scheme. In addition, we applied Poincaré mapping for identifying the limit cycles in the systems using different interpolation schemes. We considered known examples with four limit cycles and successfully reproduced them.

We first implemented the code sequentially in MATLAB, and later developed a parallel implementation in Julia using the library Cuda.jl. We benchmarked the accuracy of the solver in the analytically solvable case of a circular harmonic oscillator system. After that, we were able to accurately replicate the example provided in [1]. The developed code is freely accessible in repository hosted in `github.com/FiberFranch/Hilberts-16/` and is useful for finding the limit cycles of any quadratic ODE system efficiently using GPUs.

Resumen

El objetivo principal de este trabajo de fin de grado es abordar numéricamente la segunda parte del famoso problema 16 de Hilbert, en particular la detección de ciclos límite en sistemas bidimensionales de ecuaciones diferenciales ordinarias cuadráticas. Hemos desarrollado y verificado con éxito un algoritmo de alto rendimiento para abordar este problema de forma concurrente, aprovechando las capacidades computacionales de las unidades de procesamiento gráfico (GPU) modernas.

En este proyecto hemos estudiado el rendimiento y la precisión de diferentes métodos numéricos para resolver el sistema de ecuaciones diferenciales, y hemos elegido el mejor algoritmo con los parámetros óptimos para obtener la mejor convergencia y rendimiento. Hemos desarrollado y programado un solucionador numérico basado en el esquema de integración Runge-Kutta de cuarto orden. Además, hemos aplicado el mapeo de Poincaré para identificar los ciclos límite en los sistemas utilizando diferentes esquemas de interpolación. Consideramos ejemplos conocidos con cuatro ciclos límite y los reprodujimos con éxito.

Primero implementamos el código de forma secuencial en MATLAB, y posteriormente desarrollamos una implementación paralela en Julia utilizando la biblioteca Cuda.jl. Comprobamos la precisión del solucionador en el caso analítico de un sistema de osciladores armónicos circulares. Después de eso, fuimos capaces de replicar con precisión el ejemplo proporcionado en [1]. El código desarrollado es de libre acceso en el repositorio alojado en `github.com/FiberFranch/Hilberts-16/` y es útil para encontrar los ciclos límite de cualquier sistema ODE cuadrático de forma eficiente utilizando las GPU.

Resum

L'objectiu principal d'aquest treball de fi de grau és abordar numèricament la segona part del famós problema 16 de Hilbert, en particular la detecció de cicles límit en sistemes bidimensionals d'equacions diferencials ordinàries quadràtiques. Hem desenvolupat i verificat amb èxit un algorisme d'alt rendiment per a abordar aquest problema de manera concurrent, aprofitant les capacitats computacionals de les unitats de processament gràfic (GPU) modernes.

En aquest projecte hem estudiat el rendiment i la precisió de diferents mètodes numèrics per a resoldre el sistema d'equacions diferencials, i hem triat el millor algorisme amb els paràmetres òptims per a obtenir la millor convergència i rendiment. Hem desenvolupat i programat un solucionador numèric basat en l'esquema d'integració Runge-Kutta de quart ordre. A més, hem aplicat el mapatge de Poincaré per a identificar els cicles límit en els sistemes utilitzant diferents esquemes d'interpolació. Considerem exemples coneguts amb quatre cicles límit i els vam reproduir amb èxit.

Primer implementem el codi de manera seqüencial en MATLAB, i posteriorment desenvolupem una implementació paral·lela en Julia utilitzant la biblioteca Cuda.jl. Comprovem la precisió del solucionador en el cas analític d'un sistema d'oscil·ladors harmònics circulars. Després d'això, vam ser capaços de replicar amb precisió l'exemple proporcionat en [1]. El codi desenvolupat és de lliure accés en el repositori allotjat en `github.com/fiberfranch/hilberts-16/` i és útil per a trobar els cicles límit de qualsevol sistema ODE quadràtic de manera eficient utilitzant les GPU.

Contents

List of Figures

List of Tables

List of abbreviations

- CPU: Central Processing Unit.
- GPU: Graphics Processing Unit.
- HPC: High-Performance Computing.
- ODE: Ordinary Differential Equation.
- RK4: Fourth-order Runge-Kutta integration method.
- RKF45: Runge-Kutta-Fehlberg integration method.

1 Context

This project is a Bachelor's thesis in Informatics Engineering for the specialisation in Computer Sciences from the Facultat d'informàtica de Barcelona (FIB) of the Universitat Politècnica de Catalunya (UPC). It was carried out under the supervision of Dr. Grigori Astrakharchik and Dr. Marina Gonchenko.

In this project we took advantage of the computational capabilities of graphical processor units (GPUs) to concurrently tackle Hilbert's 16th problem through numerical methods.

1.1 Terms and concepts

1.1.1 High-Performance Computing and HPC tools

High performance computing (HPC) is one of the most essential tools fueling the advancement of computational science. HPC most generally refers to the practise of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation to solve large problems in science, engineering, or business. [2]

CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA for general computing on its own GPUs. CUDA has allowed developers to speed up compute-intensive applications by harnessing the power of general purpose GPUs to parallelise their computations. Thanks to these advancements, highly complex problems that were once impossible to compute in realistic time frames can now be computed even on average consumer hardware GPUs. [3]

1.1.2 Limit cycles

For this thesis, it was important to understand the mathematical concept of limit cycles.

In mathematics, in the study of dynamical systems with two-dimensional phase space, a limit cycle is a closed trajectory in phase space. It is topologically distinguished from neighbouring trajectories that are not closed. In two-dimensional systems, these neighbouring trajectories spiral either toward or away from the limit cycle, either as time approaches infinity or negative infinity. Limit cycles are inherently nonlinear phenomena and thus cannot occur in linear systems. [4, 5]

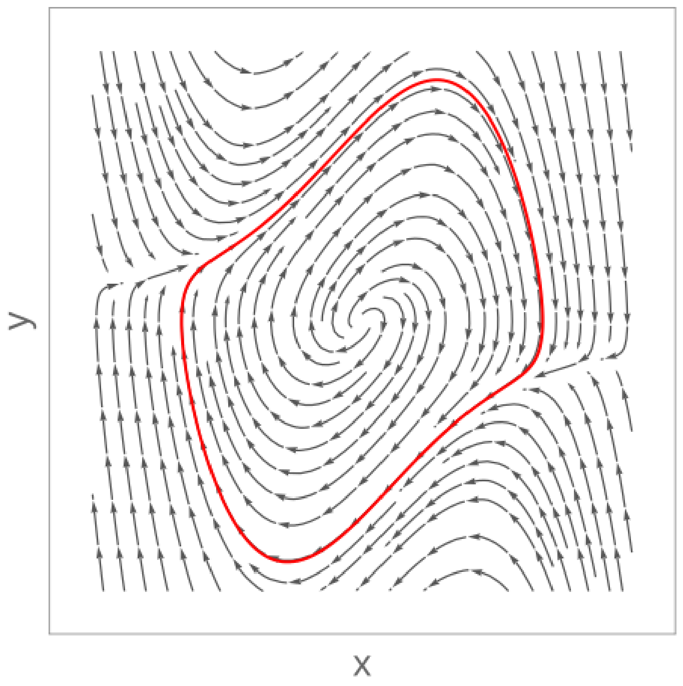


Figure 1: Example of a Van der Pol stable limit cycle in the phase space (limit cycle marked in red). [6]

The limit cycle is stable (or attracting) if all neighbouring trajectories approach it. If otherwise, all neighbouring trajectories are away from a limit cycle, it is said to be unstable. A limit cycle can be neither stable nor unstable, and in such cases it is said to be half-stable. These types are shown in Figure 2:

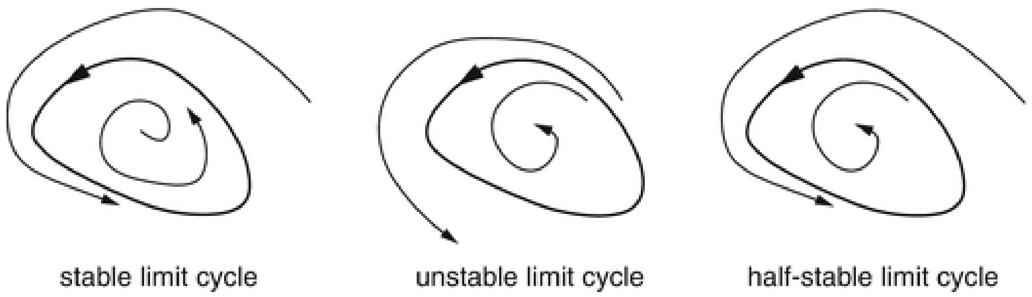


Figure 2: Typology of limit cycles. Limit cycles inidcated by bold, closed cycles. The tendency of the other trajectories indicated by the arrows. [5]

1.2 Introduction and context

1.2.1 Hilbert's Problems

In his famous lecture of the 1900 International Congress of Mathematicians, David Hilbert published a list of 23 problems with deep significance for the advance of mathematical science. These problems, ranging greatly in topic and precision, were the subject of intensive research throughout the 20th century, and even today they are the object of discussion. [7, 8]

In the 120 years since Hilbert's talk, the 23 problems have all received significant attention. By 2012, of the 23 problems, ten were considered solved, and a further seven have solutions that have partial acceptance, although there exists some controversy as to whether they are solved or not. Of the six remaining problems, three of them were considered unresolved, and three others considered too vague to even be resolved. [7, 9]

Still, some of Hilbert's problems concern what are now flourishing mathematical sub-disciplines, like the theories of quadratic forms and real algebraic curves. One such problem, and the concern of this Bachelor's thesis, is Hilbert's 16th problem. [9]

1.2.2 Hilbert's 16th Problem

Hilbert's 16th is a two-part problem, sometimes called the 'Problem of the topology of algebraic curves and surfaces'. It is one of the few problems which is still unresolved, though results on this problem continue to be published to this date. [8, 10, 11]

First part of Hilbert's 16th Problem

The first part asks for the relative positions of closed ovals of an algebraic curve given by the set of points which are solutions of a polynomial equation:

$$P(x, y) = 0$$

In 1876 Harnack investigated algebraic curves in the real projective plane and found that the maximal number of separate connected components a curve of degree n could have been no more than:

$$\frac{n^2 - 3n + 4}{2}$$

Furthermore, he showed how to construct curves that attained that upper bound, and thus that it was the best possible bound.

Regarding the relative positions, even if this is a purely algebraic problem, there has been little progress on the general case, while there is progress for small values of the degree of the polynomial P (degree less than or equal to 7). [8, 12]

Second part of Hilbert's 16th Problem

The second part of the problem is a problem of ordinary differential equations (ODEs), but with the components of the vector field given by polynomials as follows:

$$\frac{dx}{dt} = P(x, y), \quad \frac{dy}{dt} = Q(x, y)$$

This part asks for the maximal number and relative positions of limit cycles of these planar polynomial (real) vector fields of a given degree n. This problem, opened for more than a century, has been at the centre of many developments in differential equations. The main difficulty of Hilbert's problem is that, although a polynomial vector field is an algebraic object, its trajectories are not algebraic. In the neighbourhood of singular points they may not even be analytic. The fascination of Hilbert's 16th problem comes from the fact that it sits at the confluence of analysis, algebra, geometry and even logic. [8, 12]

The second part is completely open. Even if the problem was stated as early as

1900 it was only in 1987 that Ecalle and Ilyashenko proved independently that a polynomial vector field has a finite number of limit cycles.

1.2.3 Problem to be solved

The object of this Bachelor's thesis was to address the second part of the 16th Hilbert problem numerically and develop a parallel code to run on a dedicated GPU.

The principal task was to develop a highly efficient parallel code for the simulation of the 16th Hilbert problem. The goal was to develop a code capable of (i) performing the integration of a system of two differential equations with polynomials of second order and (ii) analysing the solutions for the possible presence of limit cycles.

In previous works, the maximum number of limit cycles found for vector fields of polynomial quadratic systems was 4. [1]

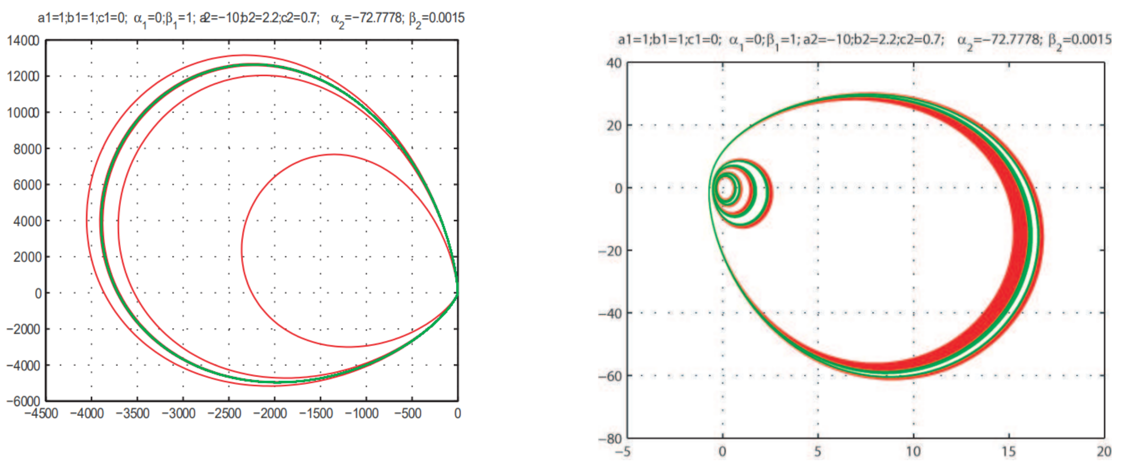


Figure 3: Visualization of four limit cycles in the phase space for a two-dimensional polynomial quadratic system. On the left, a large limit cycle in the negative half-plane. On the right, three nested normal-sized limit cycles on the positive half-plane. [1]

These vector fields of polynomials of second degree are described by the following equations:

$$\frac{dx}{dt} = a_1x^2 + b_1xy + c_1y^2 + \alpha_1x + \beta_1y$$
$$\frac{dy}{dt} = a_2x^2 + b_2xy + c_2y^2 + \alpha_2x + \beta_2y$$

Leonov [13] showed that these systems could be simplified to reduce the number of parameters to 5, obtaining the following system:

$$\frac{dx}{dt} = x^2 + xy + y$$

$$\frac{dy}{dt} = a_2x^2 + b_2xy + c_2y^2 + \alpha_2x + \beta_2y$$

Although each integration is rather simple, the phase space of the free parameters is large. If each parameter is sampled at 100 different values, a complete study requires analysis of ten billion different choices of parameters. Such an expansive task through brute-force approach would greatly benefit from a parallel approach, thus the computations will be concurrently performed on GPUs.

1.3 Stakeholders

The main stakeholders in the successful realisation of this thesis were its directors, Dr. Grigori Astrakharchik and Dr. Marina Gonchenko, and me, Francesc Forn Caabate, as the student responsible for seeing it through.

The results of this thesis may also be of use to members of the scientific community working in the field of dynamic systems, keeping in mind that this project does not provide an analytical solution to Hilbert's 16th problem, but rather offers a concurrent numerical approach from which to draw conclusions and improve computation performance.

1.4 Justification

Much work has been done on Hilbert's 16th problem; both analytical and numerical methods have been used to study the two-dimensional vector fields of quadratic polynomials. In this thesis, we focused solely on numerical approaches to the problem and how to exploit the parallelisability of the parameter-space search. Our aim was to build upon works such as Ref. [1], implementing methods to solve the ODE systems numerically to then explore the parameter-space in order to find limit cycles concurrently.

To illustrate the vastness of this problem, we offer the quote by A.N. Kolmogorov which is described by V.I. Arnold in Ref. [14]:

"To estimate the number of limit cycles of square vector fields on plane, A.N. Kolmogorov had distributed several hundreds of such fields (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 just how complex the problem we are dealing with is, and how much it stands to benefit from modern HPC. The goal was therefore to develop our own parallel code to be executed in a GPU to efficiently exhaust this parameter space.

1.4.1 Performance

The CUDA framework has API for several programming languages, including C/C++, Python, Julia, FORTRAN, MATLAB, among others. The official CUDA programming toolkit is in C/C++ and offers the most customisability and low-level configuration to adapt the code to the hardware. [15]

While performance was of the utmost importance, we also had to keep in mind the time restrictions of this thesis. For this reason, we ideally wanted to work with a programming language that allowed for fast development at high level without significant detriment to performance.

Two possibilities were the use of Python's *pyCUDA* and Julia's *Cuda.jl* 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. [15]

Another alternative was the use of MATLAB's Parallel Computing Toolbox, which allows to computationally solve data-intensive problems concurrently using high-level constructs such as parallel for-loops, special array types, and parallelised numerical algorithms.

The initial phases of the project were carried out in MATLAB due to its ease of access and the researcher's prior knowledge of the language. However, due to its limitations when parallelising the code, the program was later moved to Julia, which is, according to the literature, faster than Python, and the performance of Julia's CUDA library has only a 0.50% impact on performance compared to C. [15, 16]

1.5 Scope

1.5.1 Objectives

As presented in Section 1.2.3, the main objective of this project is to develop parallel code to simulate the 16th Hilbert problem. We aim for both correctness and performance. To this end, the following objectives and sub-objectives have been set:

Definition of the approach

Before implementing a solution, a deep understanding of the theoretical background of the thesis is required. For this task, the following points need to be covered:

- Research and review of the literature. An initial investigative phase to become familiar with Hilbert's 16th problem and the current status of its resolutions.
- Explore system dynamics and understand the concept of limit cycles.
- Study numerical methods for solving ODEs.
- Research methods for detecting limit cycles and determine their number in a given system.

With these, we will define the approach to the solution in the implementation phase.

Solution implementation

Once the theoretical bases are covered, a practical stage can begin. At this stage, the idea is to cover the following sub-objectives:

- Implement the numerical algorithms for ODE resolution and limit cycle detection in the language decided.
- Implement methods for limit cycle detection.
- Adapt code into parallel solution.

Results analysis

Once the program is developed and the results are obtained, a final phase of analysis and discussion of future steps will take place. The objectives set for this phase are as follows:

- Visualisation and analysis of results.
- Performance benchmarking.
- Study of alternatives, further exploitation of parallelisability and future work.

1.5.2 Requirements

In a thesis of this nature, where a program will be developed, the usual requirements and good practises will have to be met:

- **Readability and cleanness of code**. Following good programming practises to make the code readable, maintainable, and adaptable to the needs of other parties.
- **Correctness**. Naturally, the program must address the problem at hand in a mathematically correct manner.
- **Adequate implementation**. The implementation of the algorithms and numerical methods must take into account the convergence of methods, precision errors and other factors that will affect the results obtained.
- **Efficiency**. It is vital that the solution is developed with efficiency in mind, considering the complexity of the problem and the need for parallelisation.

1.5.3 Potential obstacles and risks

The potential risks and obstacles the ones that are usually associated with a project of this nature:

- **Deadline of the project**. The time restrictions of the project present one of the major challenges in its fulfilment. As mentioned in Section 1.2.3, there are many combinations of parameters that need to be studied, making time a precious resource. This also means that if time is not managed appropriately, the obstacles encountered may be too great to tackle within the time limit.
- **Inexperience with parallelisation**. To parallelise and execute the code on a GPU, knowledge of CUDA is required. A significant amount of time will be required to go over the teachings of the courses on parallelism and graphic cards and to determine how to apply them to the task at hand. Additionally, in order not to work with CUDA C, some time will have to be dedicated to adapt the code to a different programming language through the use of a library which adapts CUDA C to the language of choice, in this case, Julia.
- **Inexperience in the field**. My lack of expertise in mathematical fields such as system dynamics may hamper my ability to efficiently carry out this project, whether in the form of difficulties in selecting the best methods for the project, a potential inability to detect errors in the results obtained, or the slow pace at which I will be able to direct my research and read through the bibliography. With support from my tutor, I hope to overcome these.
- **Complexity of the Topic**. While some of the methods that will be employed are well known and documented, information on the topic of this thesis is in itself rather scant. That, of course, is to be expected, being a problem considered unresolved. In the documentation phase, we will study previous numerical implementations and learn from them.
- **Result accuracy and misinterpretation**. Precision errors may arise due to the use of numerical methods and limitations on hardware precision, which will need to be accounted for in order to draw adequate conclusions.

1.6 Methodology and rigour

1.6.1 Methodology

Due to the short time available for the realisation of this thesis, an *Agile*-based methodology may be best suited to organise the tasks. Following this methodology, we divide the work into tasks or *sprints* according to the different stages of development, starting with the initial implementation of the problem and from there working incrementally on the optimisation and parallelisation of the code.

1.6.2 Monitoring tools and validation

For managing and monitoring the progress of the project, we have many tools at our disposal.

- To set and manage the weekly progress and sprints, a Trello board with the updated tasks and their state will be used.
- To have access to a shared and updated project report, the files will be uploaded and edited on Overleaf.
- Lastly, for version control of the code developed we will use git. We will do this through a shared repository on GitHub. This will allow us to have access to previous versions of the code should we have the need to backtrack, to define a main branch with the current working code, and for the code to be shared and monitored by the tutor.

2 Time planning

The project started on February 21, and the expected defence project date is on June 30th. This gives us some 18 weeks to work on this project. Taking into consideration the project's credit score, it should take around 450 hours worth of work to see it through. The aim of this section is to plan out how these hours will be distributed among the different tasks.

2.1 Description of the tasks

2.1.1 Task definition

The different tasks in this project fall into four major groups:

1. Project planning and management.
2. Research.
3. Implementation.
4. Experimentation and result analysis.

Project planning and management

The aim of this planning phase is to set the course for the realisation of the project. During this phase, weekly drafts documenting different studies and approach decisions will be produced.

1. **Context and project scope**. Study of the project context and scope. Definition of the project objectives, choice of work methodology, and investigation of related works and how to take advantage of them.
2. **Time planning**. Manage and distribute available time among the different tasks to be carried out. Definition of the tasks, estimation of the time requirement for each of them, creation of a realistic schedule in which to carry out the tasks in the form of a Gantt diagram, analysis of risks and alternatives and their effect on the schedule.
3. **Budget and sustainability**. Analysis of economic and environmental impact. Identification of costs, cost estimation, management control mechanisms, sustainability report.
4. **Final document**. Last deliverable, in which all previous sections are up to date with the feedback received.

The actual documentation process will last for the whole of the project. As it is a continuous and dynamic process, some decisions or results obtained may change as we progress, so all these changes will have to be accounted for in the final document.

Lastly, there will be one more weekly managerial task, a regular meeting with the thesis director to discuss the weekly advances, check on progress, and consider where we are on the schedule.

Literature and investigation

The aim of this phase is to get a solid grasp of the many theoretical and practical topics required for this thesis. These include:

- **Read on ODE system resolution through numerical methods**. Study the many standard numerical methods for solving ODEs and devise how to apply them to the systems we are interested in.
- **Read on methods for identifying limit cycles**. Investigation on non-analytical methods for detecting limit cycles.
- **Become acquainted to MATLAB Parallel Computing Toolbox**. Study the official documentation and examples available.
- **Become acquainted to Cuda.jl**. Study official documentation. Look at practical examples and courses available online, and investigate how to apply it to project's context.

Implementation

At this stage, the idea is to carry out the more practical part of the project, mainly consisting of implementing the different algorithms, studying their potential for parallelisation, implementing the concurrent solution and running tests.

- **Implementation of of ODE system solver**. Program the methods studied for solving the systems.
- **Implementation of limit cycle detection**. Program the code responsible for exploring the parameter space sequentially.
- **Implementation of parallel solution**. Adapt the previous stages into code that can be executed concurrently in a GPU by using parallelisation strategies in Cuda.jl.
- **Code testing**. Run tests on the code to verify correctness and efficiency.

Experimentation and result analysis

Once a prototype has been developed, the trial phase may begin. In this phase, we will first execute a few tests to study the parallel behaviour of the program. After analysing the results, we will compute new cases and finally draw conclusions about the overall project.

- **Test cases**. Initial connection with GPU. Execution of test cases with known solutions to verify the correct concurrent functioning of the code.
- **Test analysis**. Performance study of the different methods comparing efficiency, accuracy, memory usage, etc.
- **Real cases**. Execution the code on new cases with the selected search space.
- **Analysis of results**. Further performance analysis and analysis of limit cycles.
- **Conclusions and further work**. Conclusions drawn from all results. Discuss possible further work.

(Note: CUDA related tasks may be considered an additional goal of the thesis, should the time allow for it. However, the MATLAB implementation and results should suffice for the goals of the project.)

2.1.2 Time estimation

In this section I will offer an estimate of the time I predict each task will potentially last.

Project planning takes place over four weeks, one week for each section. Taking into account a work load of 15 hours for each task, this is equivalent to 60 hours in total. In this phase, the duration of the weekly meetings is also included. That is, a total of 15 meetings of one hour each, so in total, this phase should encompass about 75 hours of the project.

In the **literature and investigation** phase, I expect that not much time will be needed to look into methods for solving ODEs, considering I have previously coursed subjects in numerical computing. The current estimation is that, between this and the study of limit cycles, it should take me around 35 hours. Since programming with MATLAB Parallel Computing Toolbox does not deviate much from regular MATLAB code, I estimate it should not take longer than 15 hours to become acquainted with it. However, I project I might need around 45 hours to complete a course on Julia and read documentation, as well as learn how to use Cuda.jl, given I have no prior knowledge of the language.

The **implementation** phase is the most critical of all stages of the project. I expect it to very well last over 150 hours, during which time I will adapt the numerical methods to my own solution and, most importantly, implement the parallel solutions.

Lastly, for the **experimentation and results analysis** phase, I expect around 120 hours worth of work, in which I will test the code with known cases and draw conclusions on these, then adapt it to new cases, benchmark performance, and finally analyse the results obtained. In this time, it is possible that I will need to go back to the implementation and make some changes based on the results obtained.

2.1.3 Task summary

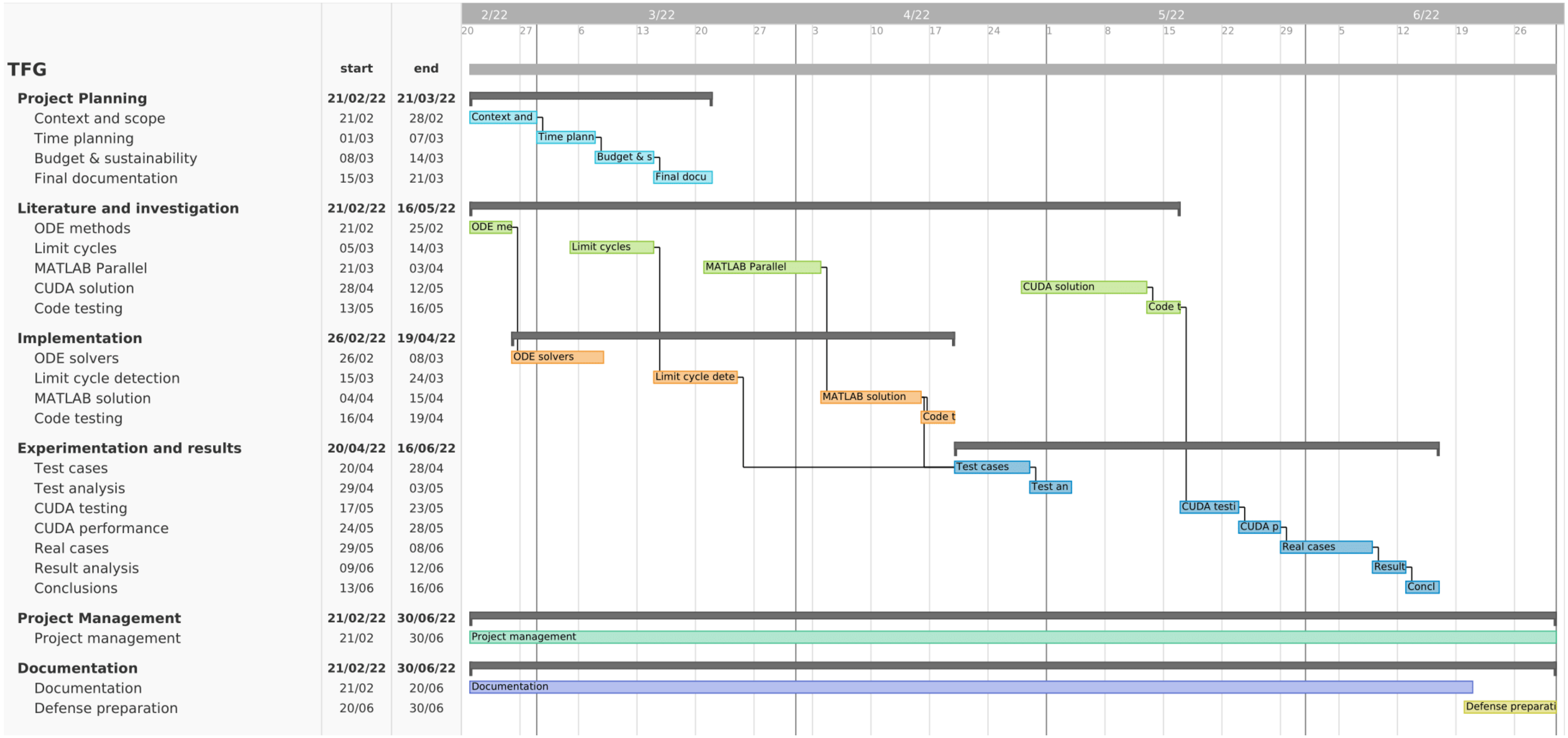
The task duration and dependencies are summarised in Table 1:

ID	Name	Prerequisites	Duration (h)
P	**Project planning**		**75**
P1	Context and scope		15
P2	Time planning	P1	15
P3	Budget and sustainability	P2	15
P4	Final documentation	P3	15
P	Meetings		15
R	**Literature and investigation**	**P**	**95**
R1	Read on ODE methods		10
R2	Read on limit cycles	R1	25
R3	Study MATLAB		15
R4	Study Julia + Cuda.jl		45
I	**Implementation**	**P**	**150**
I1	ODE solvers	R1	15
I2	Limit cycle detection	R2, I1	35
I3	MATLAB implementation	R3, I2	40
I4	Cuda.jl implementation	R3, I2	40
I5	Code testing	I3, I4*	20
E	**Experimentation and results**	**I**	**120**
E1	Test cases		30
E2	Test analysis	E1	10
E3	Real cases	E2	50
E4	Result analysis	E3	20
E5	Conclusions and further work	E4	10
M	**Project management**		**-**
D	**Documentation**		**-**
O	**Defense preparation**		**10**

Table 1: Summary of task dependencies and expected times.

2.1.4 Gantt chart

Figure 4 shows the expected schedule for the thesis in the form of a Gantt chart:



TFG	start	end
Project Planning	**21/02/22**	**21/03/22**
Context and scope	21/02	28/02
Time planning	01/03	07/03
Budget & sustainability	08/03	14/03
Final documentation	15/03	21/03
Literature and investigation	**21/02/22**	**16/05/22**
ODE methods	21/02	25/02
Limit cycles	05/03	14/03
MATLAB Parallel	21/03	03/04
CUDA solution	28/04	12/05
Code testing	13/05	16/05
Implementation	**26/02/22**	**19/04/22**
ODE solvers	26/02	08/03
Limit cycle detection	15/03	24/03
MATLAB solution	04/04	15/04
Code testing	16/04	19/04
Experimentation and results	**20/04/22**	**16/06/22**
Test cases	20/04	28/04
Test analysis	29/04	03/05
CUDA testing	17/05	23/05
CUDA performance	24/05	28/05
Real cases	29/05	08/06
Result analysis	09/06	12/06
Conclusions	13/06	16/06
Project Management	**21/02/22**	**30/06/22**
Project management	21/02	30/06
Documentation	**21/02/22**	**30/06/22**
Documentation	21/02	20/06
Defense preparation	20/06	30/06

Figure 4: Estimated Gantt chart for the project.

2.2 Resources

Human resources

For the most part, this project is carried out by me, Francesc Forn Cañabate, with the supervision and guidance of my thesis directors, Dr. Grigori Astrakharchik and Dr. Marina Gonchenko; as well as Javier Juan Morales Sorolla, who provides feedback on the drafts produced in the planning phase.

Material resources

The material resources needed for the project are mainly the bibliography, software and hardware used. These include:

- **Bibliography**: The previous papers, books, lectures, courses, etc. used to produce the solution (see References).
- **Overleaf**: The site for the production and sharing of documentation.
- **GitHub**: Where the code repository is hosted.
- **Programming languages**: Mainly Julia and MATLAB.
- **GPU**: The NVIDIA Titan V GPU in which the code will be run, and my own GeForce GTX 1070.
- **Computers**: My own PC and laptop in which the documentation and code are produced.

2.3 Risk management

As mentioned in Section 1.5.3, there exist some potential risks that can delay or even hinder the progress of this thesis. In this section we will discuss how to plan ahead of these risks, and discuss possible alternatives in case they become too obstructive.

2.3.1 Project deadline

One of the main risks in this thesis is to underestimate the work needed to see it through. This may render the predicted schedule outdated and result in delays in the solution.

- **Impact:** Medium.
- **Plan:** In the previous section, a road map was designed to plan out the different phases of the thesis. Sticking to this schedule aims to minimise the probability of delays and misuse of time.

2.3.2 Inexperience with languages

Another minor risk is the possible lack of experience of the programming languages to be used (mainly with Julia).

- **Impact:** Low.
- **Plan:** This is already accounted for in the time estimation of tasks. Julia is a popular and well documented language, so any doubt I may have is probably already solved online. Additionally, Julia has a wide range of libraries available, which may facilitate much of my work.
- **Alternative:** an alternative has already been proposed, which consists of developing a solution entirely in MATLAB, which I am already familiar with.

2.3.3 Inexperience with CUDA

Much of the difficulty of this project may arise from the development of a parallel implementation of the code using CUDA. This platform requires specific knowledge and tools which I am not acquainted with. Familiarising myself with these may be lengthy and not realisable within the duration of the project, if combined with other obstacles.

- **Impact:** Medium.
- **Plan:** The current plan already accounts for the difficulty of CUDA. We have already proposed the alternative not to work directly with CUDA C but rather to work with a library that binds the CUDA framework to a higher-lever language in order to facilitate its implementation. Additionally, CUDA-related tasks make up a large portion of the project with the purpose of trying to plan ahead of these difficulties.
- **Alternative:** Again, an alternative has been found in the form of employing the MATLAB's Parallel Computing Toolbox.

3 Budget

In this section, we will discuss the economic needs of the project.

3.1 Personnel costs

Though this project has been carried out by the thesis directors and I, the researcher, in this section we will consider the costs that would accompany the realisation of this very project in the hands of the usual members in a software development company.

In the development of a program, we would typically have the following.

- A project manager in charge of the whole process.
- The junior researcher in charge of studying the prior work, the knowledge o system dynamics and the mathematics required for the project.
- The junior developer who implements the parallel solution.
- An analyst to draw conclusions from the data.
- The tester who will verify the program's correctness.

Considering the average pay per hour for each of these roles and considering the estimates of time expenditure discussed in Section 2.1.2, we can approximate the total staff spending required. These are summarised in Table 2 and Table 3.

Role	ID	Wage (€/h)
Project manager	M	23
Junior researcher	R	15
Junior developer	D	22
Analyst	A	13
Tester	T	8

Table 2: Average wages of the development team.

Activity	Roles	Duration (h)	Cost (€)
Project planning		**75**	
Preface documentation	M	15	345
Time management plan	M	15	345
Budget and sustainability reports	M	15	345
Final documentation	M	15	345
Meetings	M, R, D	15	900
Research			
Research ODE methods	R	10	150
Research limit cycles	R	25	375
Learn MATLAB	D	15	330
Learn CUDA	D	45	990
Implementation		**130**	
ODE solvers	D	15	330
Limit cycle detection	D	25	550
MATLAB solution	D	40	880
CUDA solution	D	40	880
Code testing	T	20	160
Experimentation and results			
Test cases	D	30	660
Test analysis	A	10	130
Real cases	D	50	1100
Result analysis	A	20	260
Conclusions and further work	M, A, R	10	510
Total			**9585**

Table 3: Estimated personnel cost of activities.

And so, the expected total staff costs of the project add up to 9585€.

3.2 General costs

3.2.1 Amortisation of resources

The direct costs of this project come from the use of hardware and software licences. To consider the costs of hardware usage, we would need to take into account its amortisation.

Effectively, this thesis has been completely carried out in my home computer. Considering average of 3.5 hour workdays over a period of a little over 18 weeks, these add up to 450 hours in total. Equation 1 shows how to evaluate the amortisation:

$$Amortisation = Price \times \frac{1}{Years\ of\ use} \times \frac{1}{Days\ of\ work} \times \frac{1}{Hours\ per\ day} \times Hours\ \ used \tag{1}$$

Considering a lifespan of 5 years, 250 work days a year, 8 hours of usage per day, and the duration of 450 hours of this project, we obtain the following amortisation cost:

Hardware	Cost (€)	Amortisation (€)
Home computer	1800	81

Table 4: Hardware amortisation cost.

3.2.2 Software licencing

Another cost to consider is the price of software licencing. The only non-open source software used in this project is MATLAB. While provided by the university, its costs must also be considered for this project. A non-perpetual MATLAB license costs 800€ per year. Since this project takes place from February until June, five twelfths of the price should be covered in the project's budget, so 333.33€ in total. **All the other resources used are open source, and therefore incur no additional costs**.

3.2.3 Indirect costs

The major source of costs in this project is the use of the computer. The indirect costs associated with its usage are the Internet and electricity.

The monthly **internet cost** is around 90€/month. As seen in the previous section, if we consider that this project spans from February until the end of May, this accounts for four month of internet payments. Given the bound of 3.5 hours of work per day, we obtain a total of:

$$(90 \text{ €/month}) * (5 \text{ months}) * (30 \text{ days}/30) * (3.5 \text{ work hours}/24) = 65.63\text{€}$$

The **electricity costs** can be approximated by calculating the hours worked multiplied by the (current) average cost per kWh. Approximately, my desktop computer consumes a load wattage of around 327W. [17]

Over the course of this projects, this means a total consumption of:

$$(327 \text{ W}) * (1\text{kW}/1000\text{W}) * (450 \text{ hours}) = 147.15 \text{ kWh}$$

Taking into account the price of 0.41194 €/kWh today [18], the total adds up to 60.62€

3.2.4 General costs summary

Considering the 81€ of hardware amortisation, the 333.33€ of software licensing, and the 126.25€ of indirect costs, the total general costs add up to 540.58€

3.3 Contingency

It is a good measure to set aside a contingency margin in case unforeseen events hinder the project. We assume that a 15% of the total costs will make for an adequate budget. Therefore, considering the 9585€ of the costs per action, and the 540.58€ of general costs, we should set aside some 1518.84€.

3.4 Incidental costs

Another good practice is to take into account the costs that may arise from the risks identified in Section 2.3. We can categorise them into the risk of failing to deliver within the project deadline, and the risks arising from my inexperience.

Incident	Estimated cost (€)	Likelihood (%)	Cost (€)
Project deadline	450	25	112.5
Inexperience	400	50	200
Total	**850**	**-**	**312.5**

Table 5: Estimation of incidental costs.

3.5 Budget summary

Activity	Cost (€)
Personnel	9585
General	540.58
Contingency	1518.84
Incidental	312.5
Total	**11956.92**

Table 6: Budget summary.

3.6 Management control

Though we have discussed in previous sections how we account for potential risks and incidents in our initial budget, we also will need to model how the actual costs deviate from our estimations. Upon completing a task *t*, we will calculate the budget deviation as follows:

$$d_t = E_t - R_t \tag{2}$$

Where:

- $\mathrm{E_t}$ = **Estimated cost** of the task *t* in the initial budget.
- $\mathrm{R_t}$ = **Real cost** of the task *t* upon completion. The budget cost estimates will have to be recalculated for this.
- $\mathrm{d_t}$ = **Budget deviation** of task *t*.

A negative deviation indicates that the budget for a task was underestimated, and the funding for it will have to be obtained elsewhere. On the other hand, a positive deviation indicates that the cost of the task was overestimated, and the leftover budget can be allocated elsewhere, for instance to make up for those negative deviations, or to a fund in case of further risks or incidences.

If this were an actual project, with the employees discussed, and such, it would be good practise to keep a tab on the deviations in the form of a spreadsheet to continuously evaluate the budget situation, to easily detect the need for further funding or discuss the possible applications of unused resources.

4 Sustainability assessment

4.1 Self-assessment

As this is not my first engineering thesis of this nature, I like to think I am well acquainted with some of the concepts surrounding sustainability, both environmental, social, and economic.

As with any project, we have responsibility on how we will impact society and the environment, so it is important when planning the thesis to consider the factors described in order to maximise the positive impact.

The indicators provided and the project requirements offer useful tools to ensure that our thesis covers the baselines for sustainability. For example, numerically evaluating the economic impact or environmental footprint of hour project design may lead us to discussing and analysing possible alternatives that may lead to better performance, reduction of waste, or improve the impact our project will have both on ourselves and the people it impacts, be it directly or indirectly.

The questionnaire made me reflect not only on aspects that I was already aware of but also on some aspects that I may have otherwise not considered. Still, many of them are not reflected in this body of work, due to time restrictions, relevancy, and the scope of the thesis. Nevertheless, these aspects are important and significant and should therefore not go unmentioned or ignored.

4.2 Environmental impact

PPP: Have you estimated the environmental impact of undertaking the project?

Though not in detail, in Section 3.2.3, we saw the estimated consumption of the PC I will work on throughout the thesis. As there are no production processes, distribution channels, or other logistics involved in this thesis, electrical consumption is our main environmental concern. From the aforementioned consumption estimate, we can easily estimate the kilograms of CO_2 produced throughout the project (around 63.13 kg).

PPP: Have you considered how to minimise the impact, for example by reusing resources?

As the main consumption comes from computer usage, the main way to reduce the impact (considering working hours cannot be reduced) would be through the use of a more energy-efficient machine. Using a public PC, such as the ones available on campus, would be highly unpractical, both in terms of time efficiency and transportation emissions. Investing in a new, less consuming machine is also out of the question for many reasons, including the actual budget of 0€ for the project or the environmental impact of acquiring a new computer and recycling / discarding the old one. **Exploitation: How is the problem that you wish to address resolved currently (state of the art)?**

Currently there exist some numerical solutions to Hilbert's 16th problem. In this project we have been significantly influenced by them and aim to build on top of their work.

Exploitation: In what ways will your solution environmentally improve existing solutions?

The main objective for this thesis is to develop a solution that will be on-par with existing ones but maximising the computing efficiency. This translates not only into reduced execution times, but also into lower emissions and less resources and computer power wasted.

4.3 Economic impact

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

In Section 3 we have provided and in-depth analysis of this project's material and human needs, as well as potential risks and contingencies

Exploitation: 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?

Again, there exist some implementations of numerical solutions of the problem we want to solve. As mentioned in the previous section, this project aims to reduce both the resources and the time required to solve the problem, which translates into economic savings.

4.4 Social impact

PPP: What do you think that carrying out the project has contributed to you personally?

The realisation of this project has allowed me to implement various skills and much of the knowledge I have accumulated throughout the degree, and has given me insight into a new field that I did not know before. **Exploitation: 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 solutions? Is there a real need for the project?**

Though no apparent direct benefit is clear to me (given my lack of knowledge in the fields involved), I am convinced it this will contribute positively to the field of system dynamics that is concerned with the topics reviewed in this thesis; and could bring some insight into Hilbert 16th problem, which is still to this day considered unresolved.

5 Implementation and results

5.1 Limit cycle detection method

As stated in Section 1.2.3, the aim of this thesis was to identify the limit cycles in two-dimensional second-order differential systems. To do this, we devised a method based on Poincaré mapping.

Poincaré maps (or first return maps) are helpful for analysing systems that show periodic behaviour. A Poincaré map defines a section Σ, transverse to the flow of the trajectories in the phase space. This section is a map from Σ to itself, and what it tells us is that, if we start at any point in this map and run forward in time following a trajectory, where will we next intersect the map. That is to say, if we start at a point k in our section, a Poincaré map will tell us where the next point $k + 1$ lies on the section after one rotation through our flow.

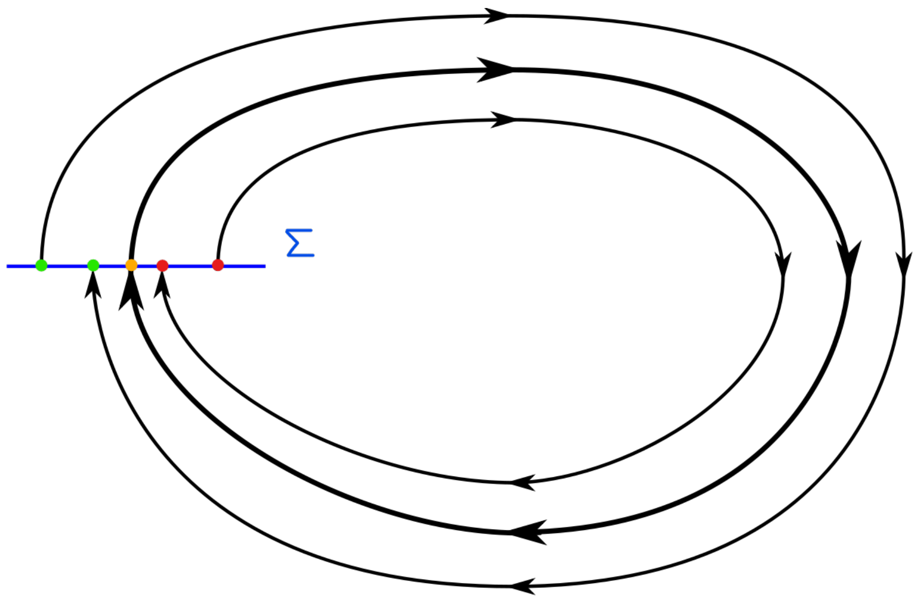


Figure 5: Illustration of Poincaré mapping for identifying limit cycles in the phase space. Starting on a point on the transverse section, Σ, we follow the trajectory for one cycle and obtain its ending point. This is how the mapping relates the initial and final points for each trajectory. The section Σ is coloured in blue. In this example, an attractive limit cycle in draw in bold, and two trajectories orbiting towards it are shown to illustrate the mapping described. [19]

This can help us identify limit cycles in the following way: if we compute one rotation starting at every point in our section, and compute the euclidean distance between the starting and ending points, we can identify limit cycles as those trajectories in our section where the distance between the starting and ending points is zero (as they are closed trajectories).

We can formalise this somewhat. We start by assigning values to the parameters a_2, b_2, c_2, α_2, and β_2. For these parameters, we solve a set of initial value problems given by each point in the section.

Generally, we took the section Σ to be the x-axis, so we defined a set of initial value problems given by the starting point $(x_0, 0)$, $\forall\ x_0 \in \Sigma$ (which is to say $\forall\ x_0 \in \mathbb{R}$, as Σ is the x-axis).

So, as described, for each value of x_0 in the section we computed the trajectory of the curve that passed through $(x_0, 0)$, understanding by trajectory one complete cycle of the segment of the curve, starting at the initial point, and passing through the x-axis **twice**, as is shown in the following Figure 6:

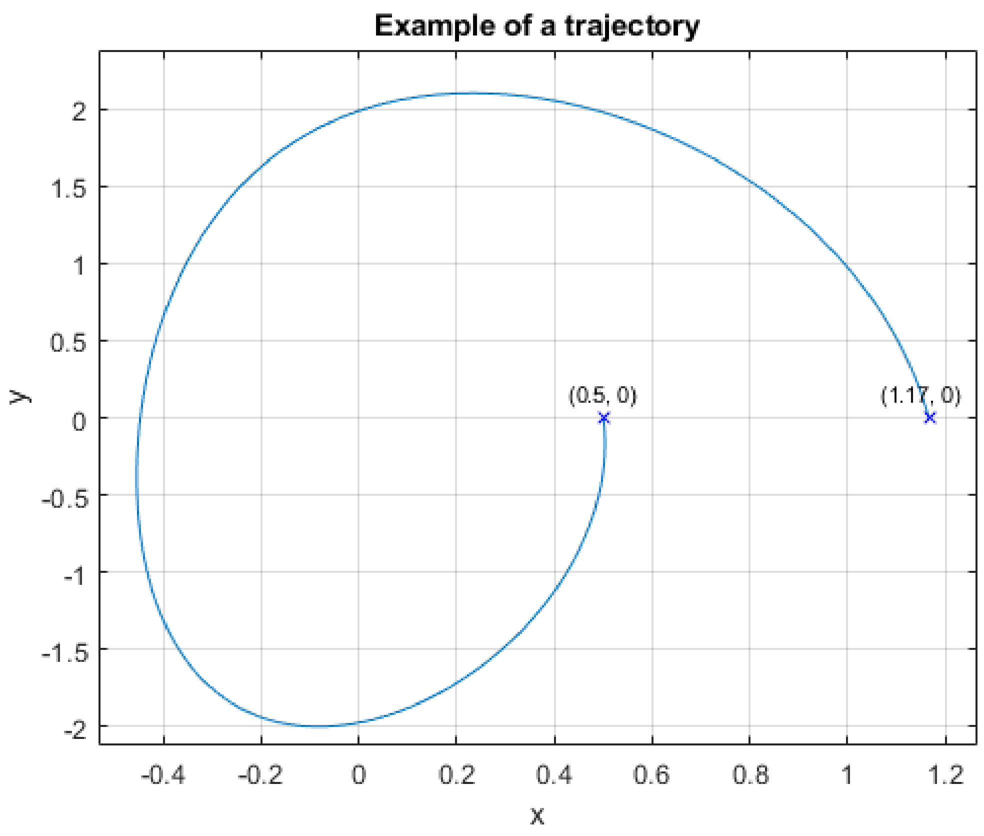


Figure 6: Example of a trajectory of in phase space. This trajectory starts at the initial point in the x-axis $(0.5, 0)$ and ends at the point $(1.17, 0)$.

After computing the trajectory, we measured the distance between the starting point and the ending point.

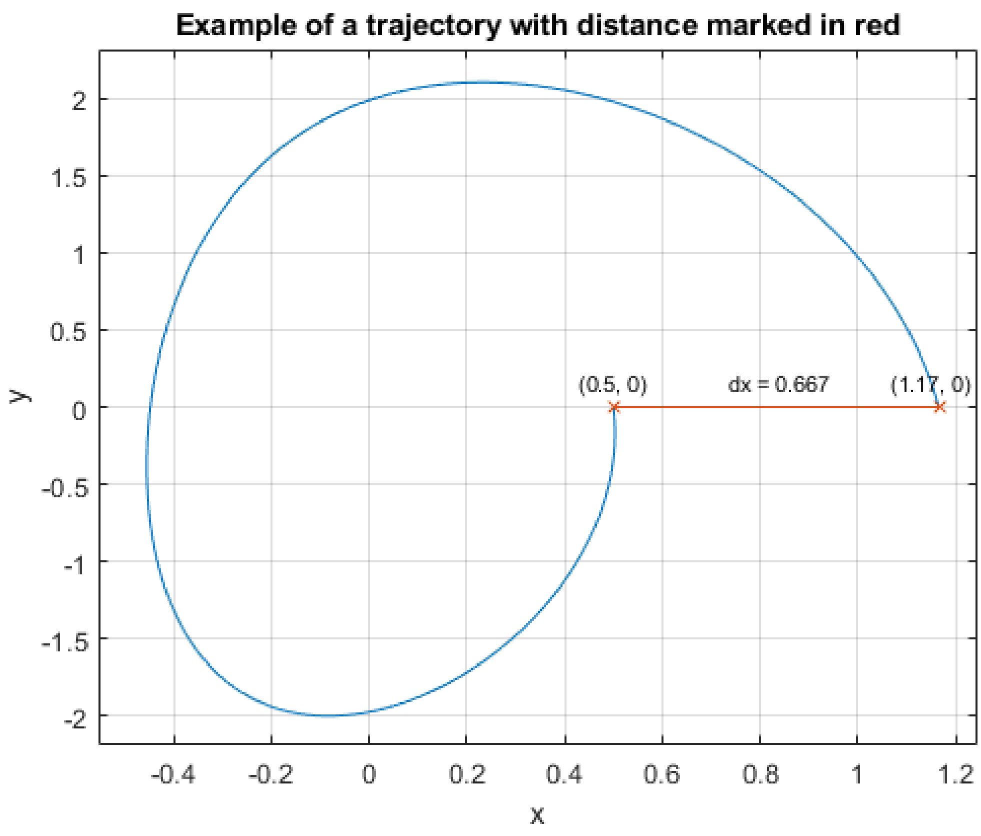


Figure 7: Representation of the same trajectory in phase space (in blue) with the distance between start and end of trajectory (in red). Total distance, dx = 0.667 is displayed.

As in the example above, we measured this distance for all the initial value problems and studied how it evolves as we move along the x-axis.

We plotted the distance against the value of x_0 to observe how the system evolved and to visualise the presence of limit cycles, as is done in Figure 10. Keeping in mind that each cycle crosses the x-axis **twice**, the number of crossings is twice the number of limit cycles.

To summarise, for the given values of the parameters, we scanned the x-axis within a given range until we found up to four cycles (as demonstrated in article [1]). With the prior knowledge that there are generally three small limit cycles in the positive half-plane, and a large limit cycle at the negative half-plane, the search space could be somewhat reduced.

In the next section, we described some numerical methods considered to integrate the ODE systems to obtain the curves.

5.2 Numerical Methods

5.2.1 Runge-Kutta 4

The first methods we considered were the Runge-Kutta methods. The Runge–Kutta methods are effective and widely used methods for solving the initial-value problems of differential equations. They can be used to construct high order accurate numerical approximation of functions, without needing their high order derivatives. [20]

Consider the following (two dimensional) first-order initial value problem:

$$\begin{cases} x' = f(x,y) \\ y' = g(x,y) \\ t_0 \leq t \leq t_f \end{cases}$$

with the initial values:

$$\begin{cases} x(t_0) = x_0 \\ y(t_0) = y_0 \end{cases}$$

The Runge-Kutta methods divide the interval $[t_0, t_f]$ into N sub-intervals $[t_i, t_{i+1}]$, for i = {0, 1, ..., N - 1}. Starting with the sub-interval $[t_0, t_1]$, with the known initial values of (x_0, y_0), the aim is to calculate the values of (x_1, y_1) by approximating the slope of the functions within the interval. Having approximated these values, the method moves on to the next sub-interval and so on.

The most widely known member of the Runge-Kutta family is the fourth-order method. It is generally referred to as “RK4”, the “classic Runge–Kutta method” or simply as “**the** Runge–Kutta method”.

In the Runge-Kutta 4 method, the values of x and y are approximated by the following equations:

$$x_{i+1} = x_i + \frac{1}{6}h(k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x})$$

$$y_{i+1} = y_i + \frac{1}{6}h(k_{1y} + 2k_{2y} + 2k_{3y} + k_{4y})$$

where

$$t_{i+1} = t_i + h$$

Here x_{i+1} and y_{i+1} are the RK4 approximations of $x(t_{i+1})$ and $y(t_{i+1})$ respectively.

At any time $t = t_i$ the values of x_{i+1} and y_{i+1} are determined by the current known values x_i and y_i plus the weighted average of four increments, where each increment is the product of the size of the interval, h, and an estimated slope specified by functions f and g.

The expressions for the slopes are the following:

$$\begin{cases} k_{1x} = f(x_i, y_i) \\ k_{1y} = g(x_i, y_i) \\ \\ k_{2x} = f(x_i + \frac{1}{2}k_{1x}, y_i + \frac{1}{2}k_{1y}) \\ k_{2y} = g(x_i + \frac{1}{2}k_{1x}, y_i + \frac{1}{2}k_{1y}) \\ \\ k_{3x} = f(x_i + \frac{1}{2}k_{2x}, y_i + \frac{1}{2}k_{2y}) \\ k_{3y} = g(x_i + \frac{1}{2}k_{2x}, y_i + \frac{1}{2}k_{2y}) \\ \\ k_{4x} = f(x_i + k_{3x}, y_i + k_{3y}) \\ k_{4y} = g(x_i + k_{3x}, y_i + k_{3y}) \end{cases}$$

where:

- k_1 is the slope at the beginning of the interval (as in Euler's method).
- k_2 is the slope at the midpoint of the interval, using the value of k_1.
- k_3 is the slope at the midpoint of the interval, but now using the value of k_2.
- k_4 is the slope at the end of the interval, using the value of k_3.

In averaging the four slopes, greater weight is given to the slopes at the midpoint. [21]

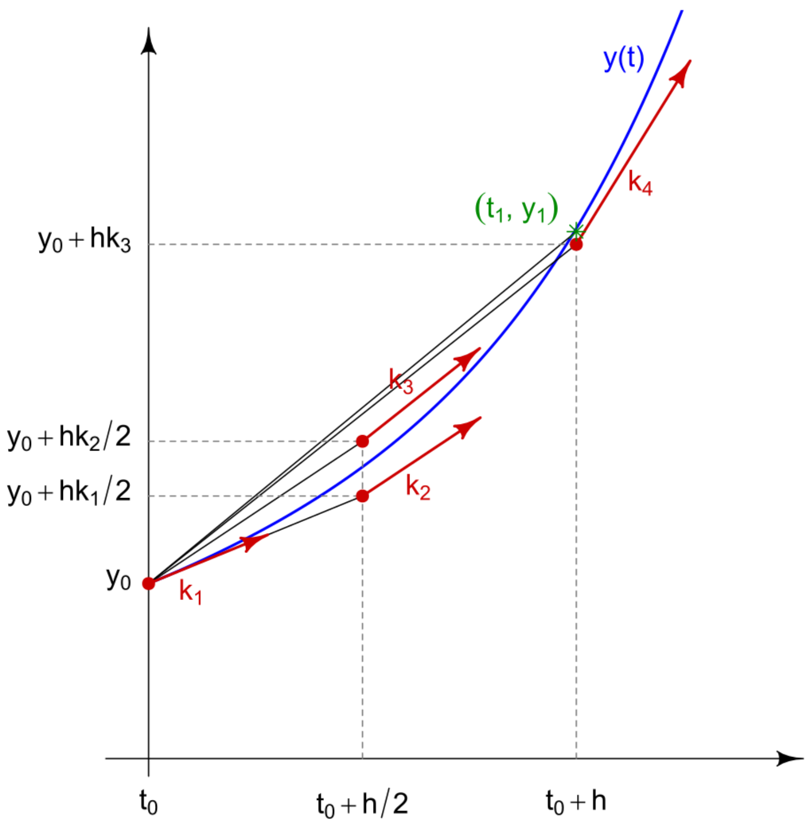


Figure 8: This image illustrates the slopes used by the Runge-Kutta 4 method for a given one-dimensional function $y(t)$, for which we know the derivative $y'(t)$, and the initial point $y(t_0) = y_0$. Here we want to evaluate the function at the time $t = t_1$. The slopes evaluated by the RK4 method are indicated by the red vectors: k_1 at the beginning of the interval, k_2 and k_3 at the midpoint, and k_4 at the end of the interval. The exact trajectory of the function in time-position space is plotted in blue, and the RK4 approximation of the point at time t_1 is depicted by the green asterisk at (t_1, y_1), which lies very close to the real value. The black lines join the initial point to each slope and to the final point. [21]

Comparison of Runge-Kutta methods

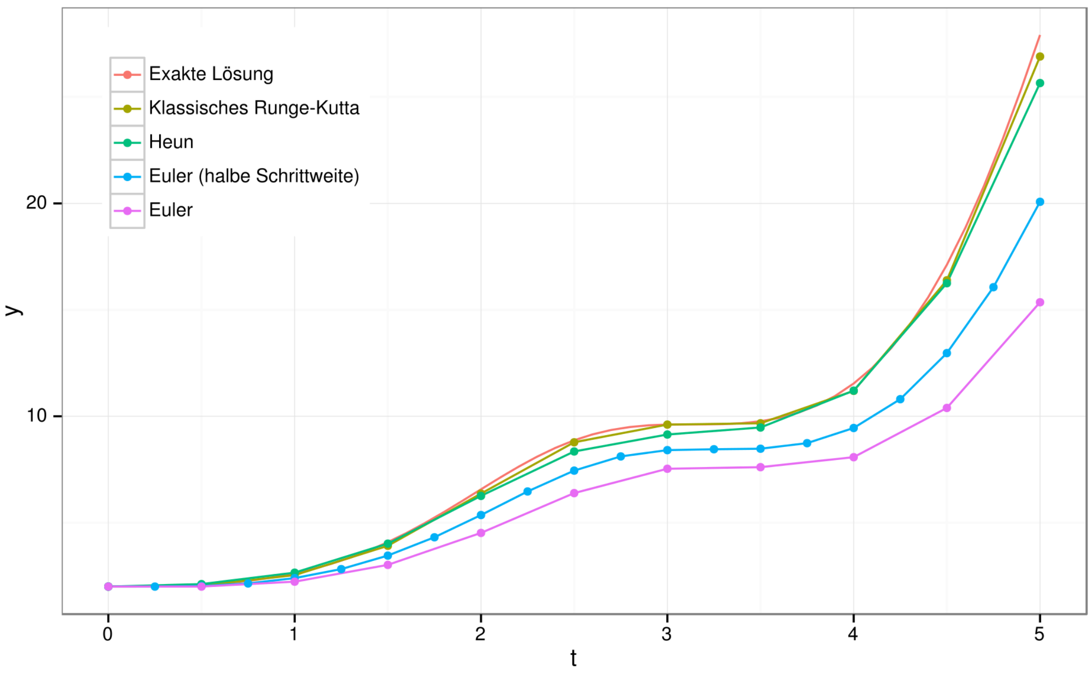


Figure 9: Comparison of the Runge-Kutta methods for the differential equation $y' = \sin(t)^2$, the exact function plotted in orange. In the figure, the trajectories y(t) correspond to numerical solutions obtained using different methods: in violet Euler's (first order) method, in blue Euler's method with the time increment reduced in half, in green Heun's (second order) method, and in olive, RK4. The plot shows that, not only is the RK fourth order method the most precise, but that it also lies very close to the real function. [21]

Figure 9 provides a comparison of the precision of different-order RK methods. As the figure shows, the results obtained through the Runge-Kutta 4 methods were more precise than those obtained by its lower-order counterparts. Additionally it shows that the RK4 method provides a an accurate approximation of the real function (provided the time step is adequate).

5.2.2 Adaptive algorithms and the Runge-Kutta-Fehlberg method

Numerical ODE solvers need to use appropriate step sizes between evaluation points to achieve high levels of accuracy while also quickly finding solutions across the evaluation interval. Adaptive ODE solvers reduce step size, h, where needed to meet accuracy requirements. To improve performance, they also increase the step size when the accuracy requirements can be satisfied with a larger step size. [22]

The Runge-Kutta-Fehlberg method (denoted RKF45) uses fourth and fifth order Runge-Kutta ODE solvers to adjust the step size at each evaluation point.

Some adaptive ODE algorithms make course changes to the step size such as either cutting it in half or doubling it. The RKF45 function first finds a scalar value, s, from a tolerance value and the difference between the solutions from the two algorithms. The new step size comes from multiplying the current step size by the scalar. The scalar will be one if the difference between the solutions is half of the tolerance. The algorithm progresses to the next step if the difference between the two solutions is less than the tolerance value. Otherwise, the current evaluation is recalculated with a smaller step size. The multiplier is restricted to not exceed two. So the step size will never increase to more than twice its current size. The tolerance and the threshold to advance to the next point are tunable parameters. [22]

5.3 Software built-in functions

Both Julia and MATLAB have their own libraries and built-in functions to integrate ODEs.

5.3.1 MATLAB's ode45

MATLAB's standard solver for ODEs is the built-in function ode45. This function implements a Runge-Kutta method with a variable time step for efficient computation. ode45 is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a single-step solver – in computing $y(t_{i+1})$, it needs only the solution at the immediately preceding time point, $y(t_i)$. [23]

5.3.2 Julia libraries

Julia has many libraries available to solve ODEs. The most prominent among them are *DifferentialEquations.jl* and *DiffEqGPU.jl*.

5.4 Methods implemented

5.4.1 Integration method used in this thesis

For the purposes of this thesis, we opted to program our own implementation of the RK4 method to solve the systems, so that we could adapt it to our needs in finding each trajectory. We implemented it first in MATLAB to run tests and evaluate performance, and later on in Julia to parallelise it using the Cuda.jl library.

5.4.2 Additional remarks on accuracy

Rarely did the previous methods evaluate the systems at the point where they intersects with the x-axis. It was, of course, more probable that they evaluate points before and after crossing it, since they use discrete time steps h (fixed or adaptive). For this reason, further methods were necessary to close in on the crossing to the desired degree.

There were many ways to do this, for instance, using one of the previous methods with a finer step in the region where the function changes sign. Another option was to use inverse interpolation methods to attempt to find this value. In Section 5.5.2 we analysed the accuracy of reverse linear interpolation, reverse quadratic interpolation, and time-step reduction.

5.5 Testing

In this section we will cover the tests performed to analyse the correctness and precision of the methods used.

5.5.1 Replication of known results

With the implementation of our integration methods, we first had to analyse their correctness. To this end, we reproduced the example provided in the paper "Vizualisation of four normal size limit cycles in a two-dimensional polynomial quadratic system" [1] (parameter values of $a_2 = -10$, $b_2 = 2.2$, $c_2 = 0.7$, $\alpha_2 = -71.22$, and $\beta_2 = 0.0015$).

Following the methodology explained in Section 5.1, we started with the positive limit cycles. We used $x \in [-1, 20]$, discretised in steps of 0.01.

With an implementation in MATLAB of the RK4 method, we produced the plot of the cycle start-to-end distance against the initial value of x, as shown in Figure 3:

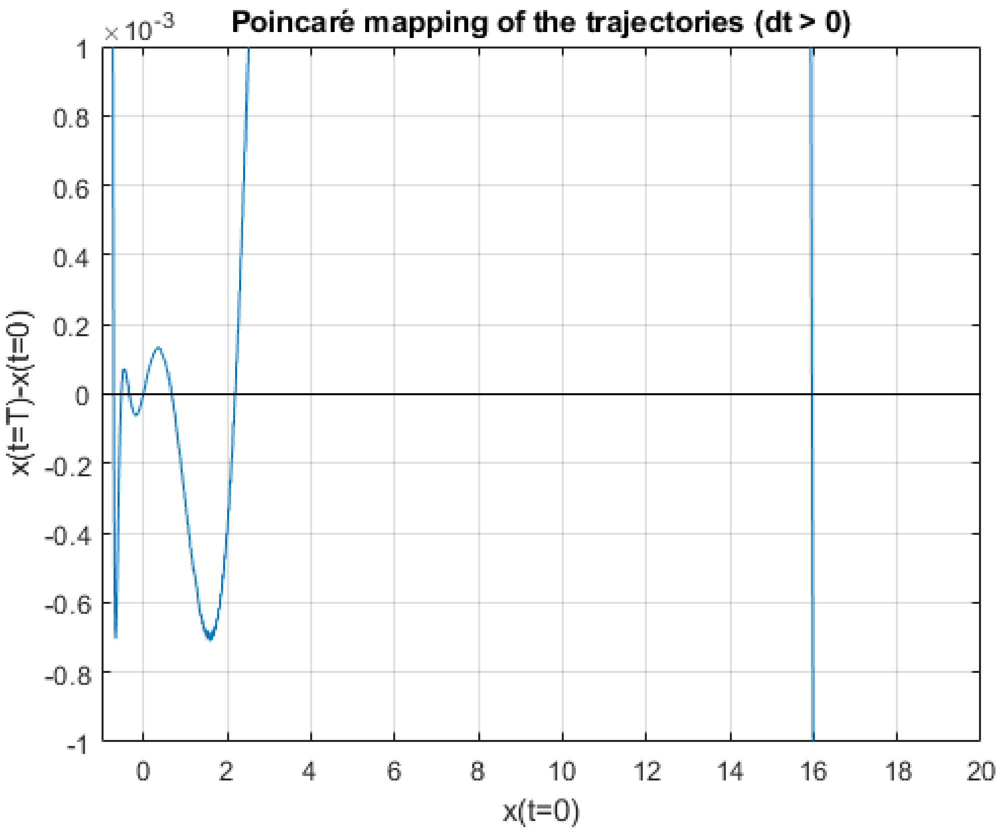


Figure 10: Poincaré mapping of the start-to-end distances of trajectories in the positive semi-axis using a positive time step. The presence of limit cycles is indicated by the x-axis crossings, two for each limit cycle.

In Figure 10 we identified the limit cycles going through the approximate points $(0.66, 0)$, $(2.2, 0)$, and $(15.96, 0)$. The other three negative intersects correspond to their other crossing to complete the cycle, remembering that each cycle crosses twice. These limit cycles are shown in the following Figure 11.

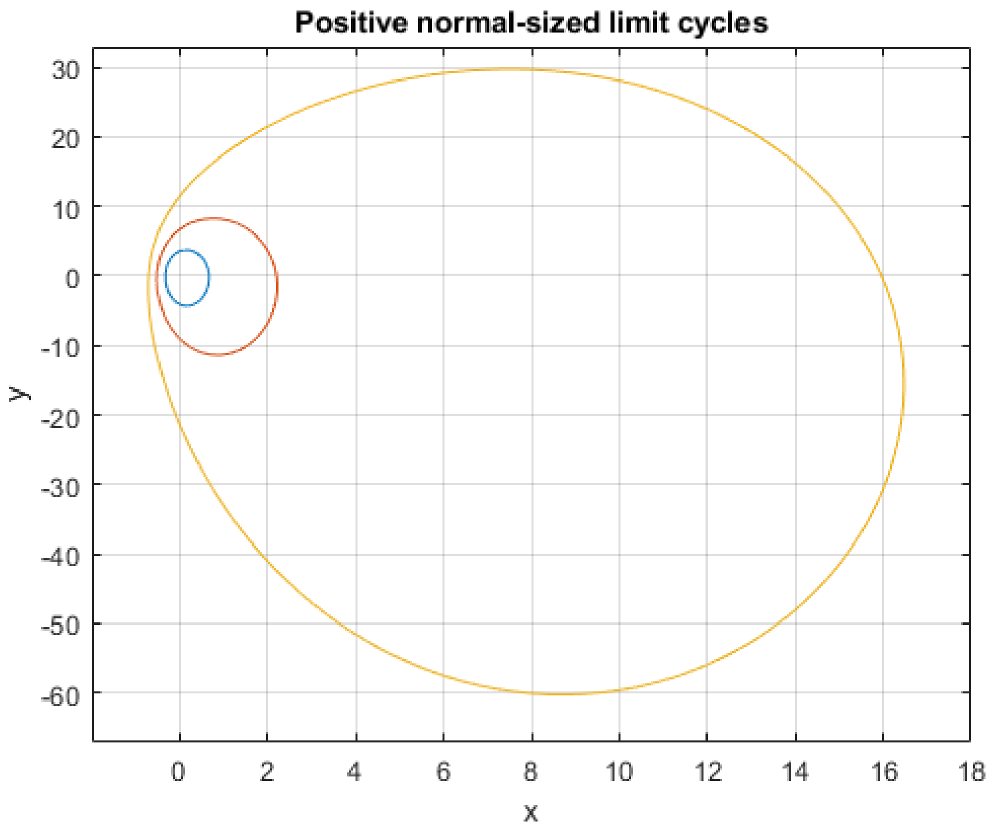


Figure 11: Example of three nested positive normal-sized limit cycles in the phase space. Parameter values taken from article [1]

The results matched those obtained in the paper with the same parameter values, so we concluded that our implementation of RK4 is on par with the paper's use of MATLAB's ode45 (and our own tests with the built-in function).

To cover our bases, we repeated this process with a negative time step to ensure we obtained the same results:

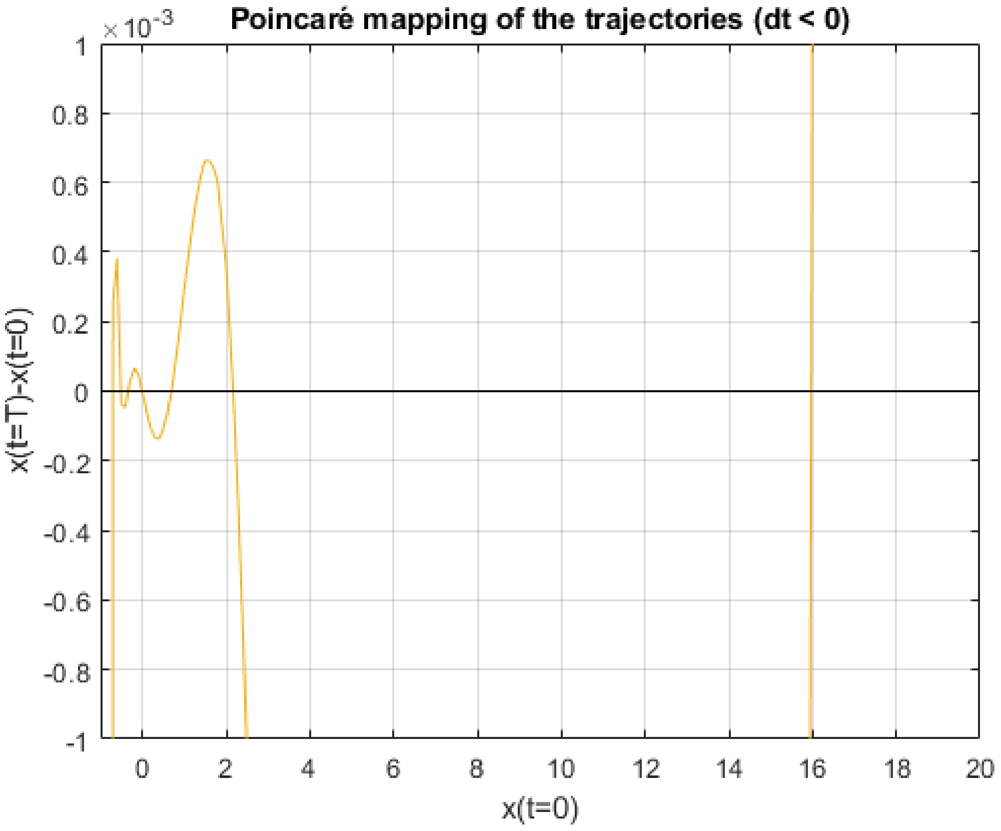


Figure 12: Poincaré mapping of the start-to-end distances of trajectories in the positive semi-axis using a negative time step. The presence of limit cycles is indicated by the x-axis crossings, two for each limit cycle.

As we expected, the crossings with both time steps were the same:

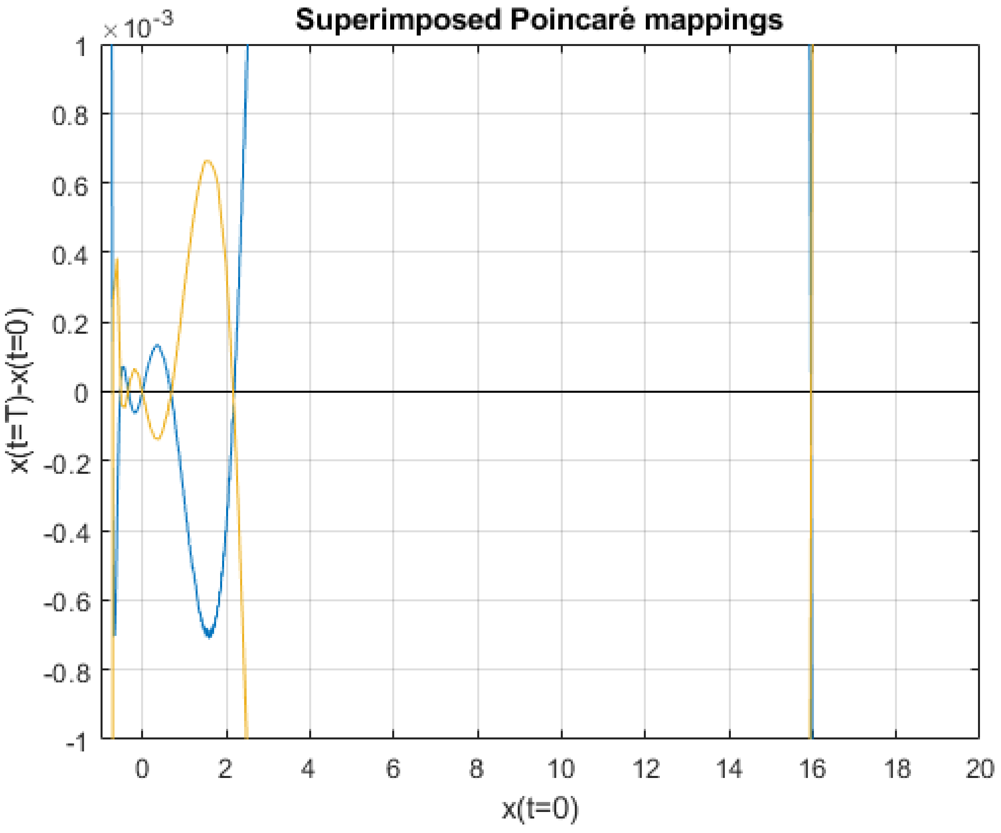


Figure 13: Superposition of Poincaré mappings of the start-to-end distances of the trajectories with positive and negative time steps. They intersect at the same x-axis crossings, indicating that the limit cycles detected with forward and backward time steps match.

Following the same process for the large negative cycle, for $x \in [-3900, -3500]$ this time with a step of 0.5, we obtained:

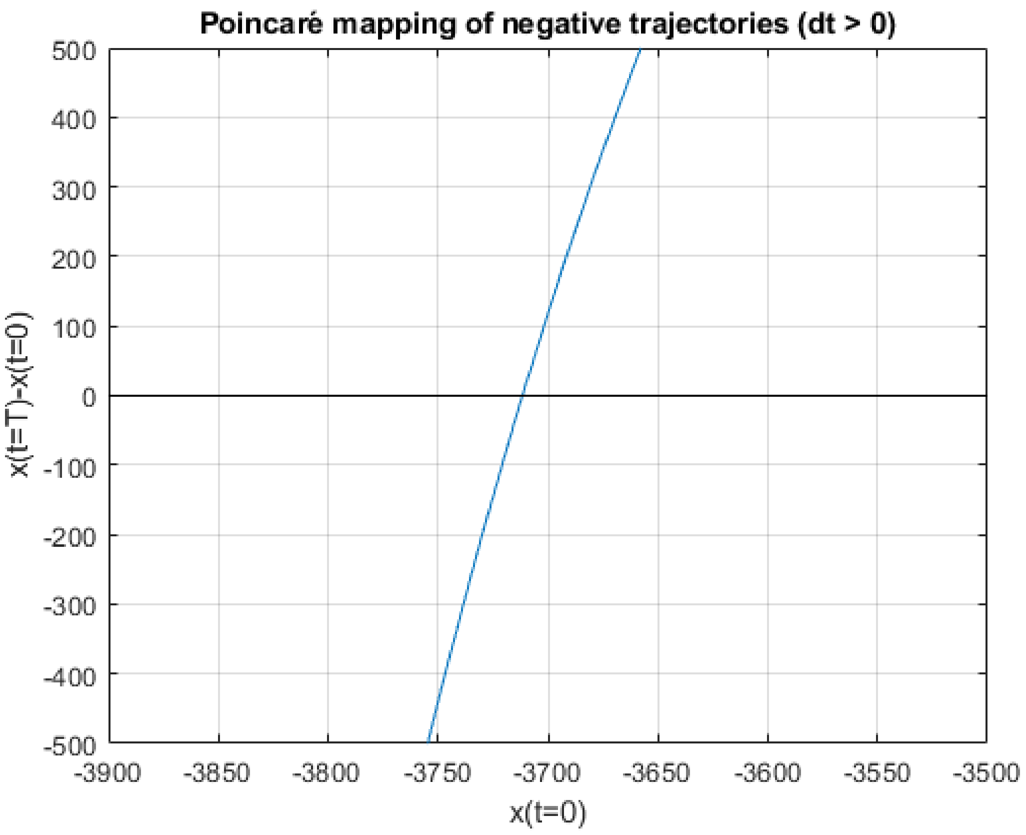


Figure 14: Poincaré mapping of the start-to-end distances of trajectories in the negative semi-axis using a negative time step. The presence of the limit cycle is indicated by the x-axis crossing. In this case, the other crossing of this cycle has been cut out to zoom-in on the region around this intersection. This is the example of the large negative limit cycle provided in [1].

This indicated the presence of the large cycle that goes through the point $(-3712, 0)$, which is shown in the following figure.

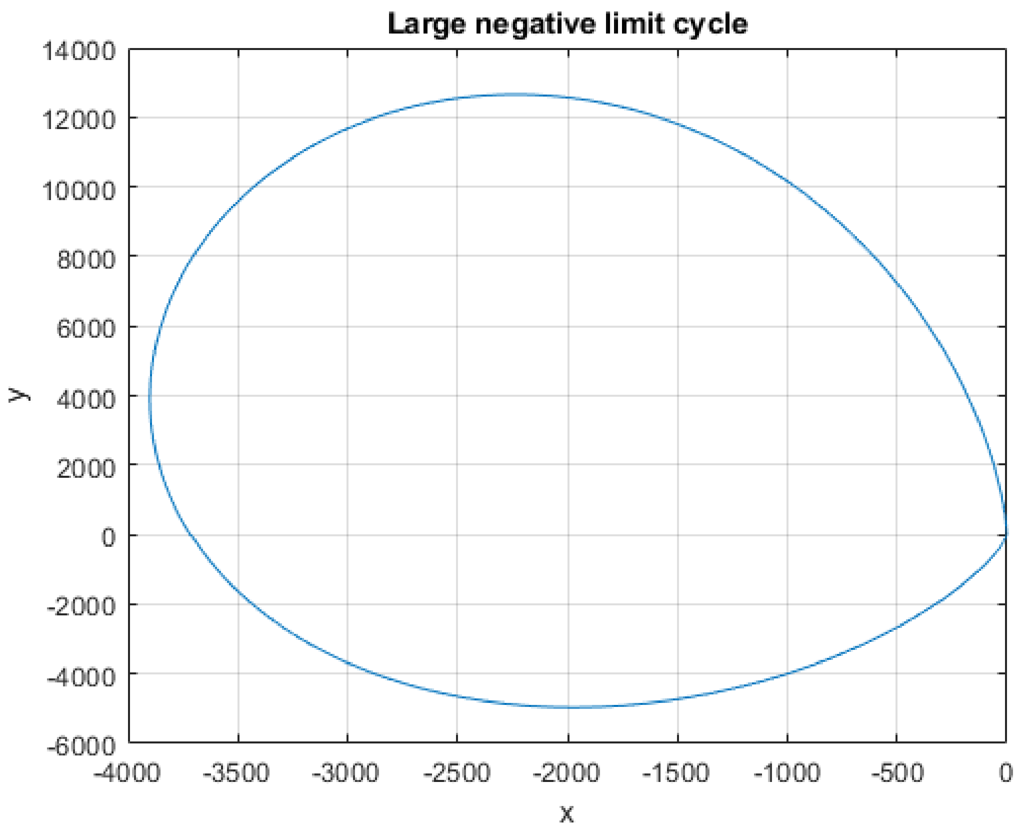


Figure 15: Example of large negative limit cycle in the phase space. Parameter values taken from article [1]

Having seen that our results matched those of [1], we believed our method to be correct and ready to be applied to further cases.

5.5.2 Precision tests

In Section 5.4.2 we discussed the need of further numerical methods so as to pinpoint the final point of each trajectory. In this section, we analysed the precision of each method.

To do this, we used a *harmonic oscillator* system in order to be able to compare the experimental results, with real well-known results.

By imposing a starting point at $(10, 0)$, the expressions that describe this system are the following:

$$\begin{cases} x' = y \\ y' = -x \\ (x_0, y_0) = (10, 0) \end{cases}$$

This yielded the known result of a circumference of radius 10 centred around the origin:

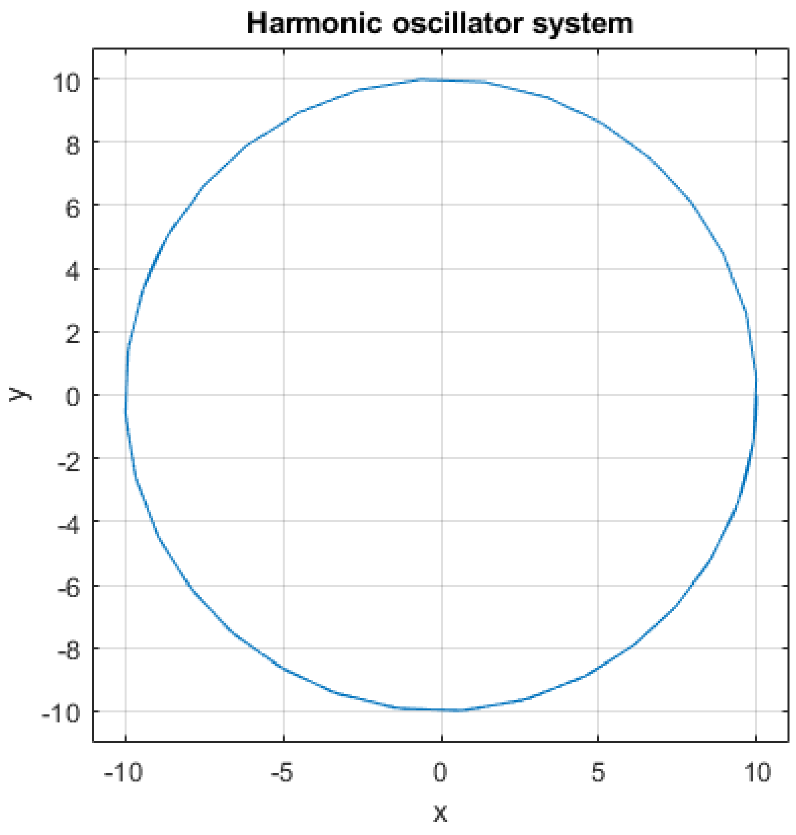


Figure 16: Solution of the harmonic oscillator system in the phase space. As mentioned, the result is a circle centred at the origin with radius of 10 units.

How these tests worked: we computed one whole cycle of the circumference with different time steps and measured how the final point approximated by RK4 combined with the numerical method of interest deviated from the final theoretical point of (10, 0). We plotted the results on a log-log plot and compared them with polynomials of different orders.

Inverse linear interpolation

First, we ran the test with inverse linear interpolation, and we obtained the following results:

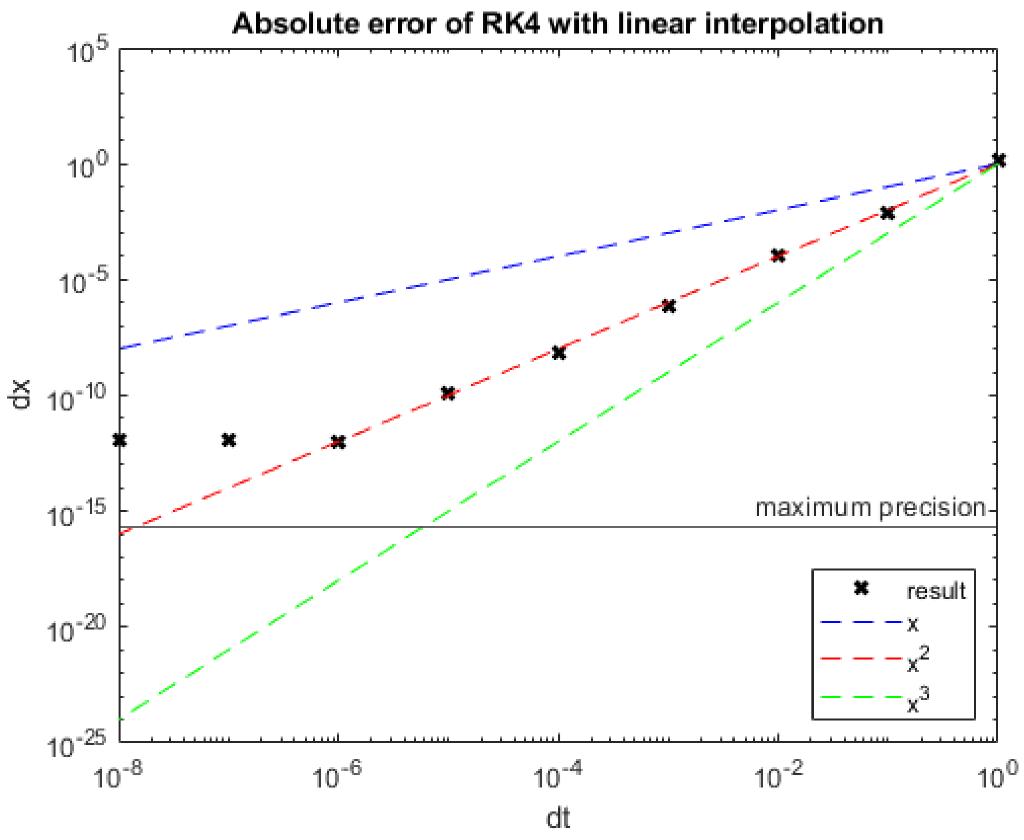


Figure 17: Absolute error of RK4 integration of the harmonic oscillator system combined with inverse linear interpolation of the crossing. The x marks correspond to the results obtained with RK4 and linear interpolation at the crossing. The dashed lines correspond to typical time step dependence of linear, quadratic, and cubic methods, which in log-log plots are represented by straight lines. The horizontal black line depicts the maximum precision obtainable, which correspondss to the machine epsilon, ϵ.

With this method the maximum precision was of the order of 1×10^{-12} when using a time step of 1×10^{-6}. At this point,point, we reached the maximum precision of this method, and any further value of dt would not improve the precision. As we can see, the order of the absolute error lies on the line of x^2.

Note: MATLAB works with double precision, so the maximum precision we could aspire to is of 2^{-52}, which is indicated with the horizontal black line labeled maximum precision in all graphs.

Inverse quadratic interpolation

Then, we ran the test with inverse quadratic interpolation and we obtained the following results:

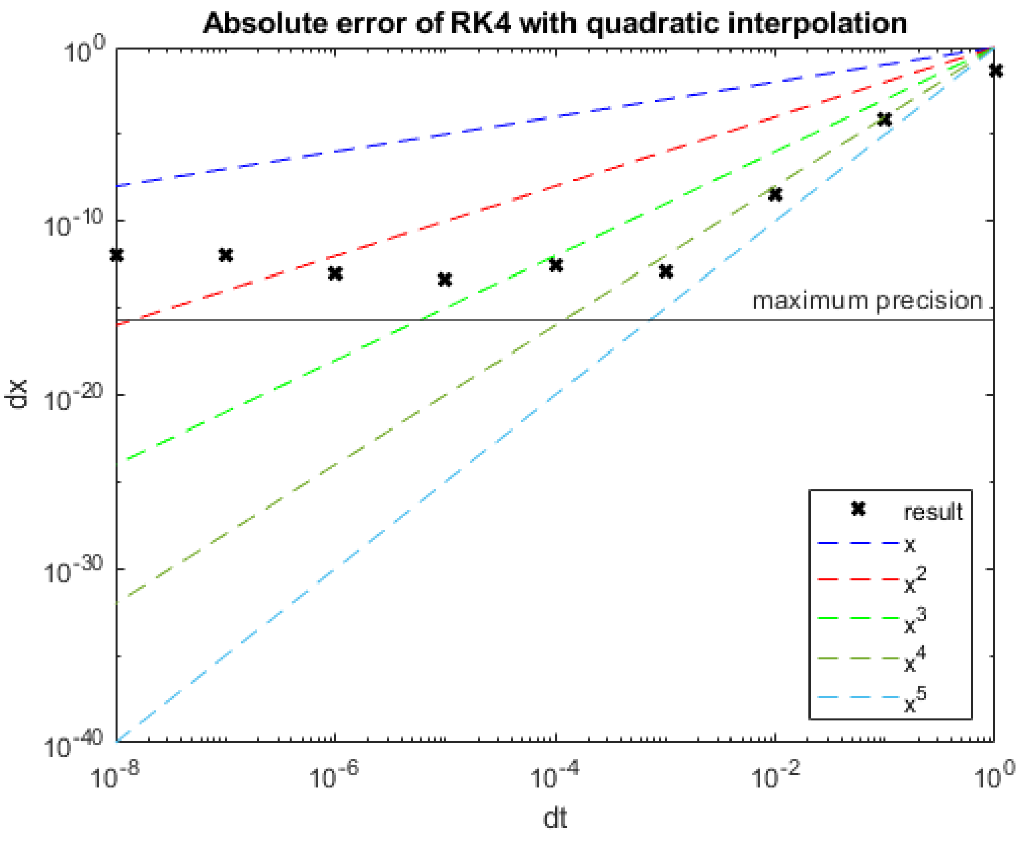


Figure 18: Absolute error of RK4 integration of the harmonic oscillator system combined with inverse quadratic interpolation of the final point.

With this method, we could achieve a precision of 1×10^{-13} with a time step of only 1×10^{-3}, which was already a great improvement over the previous method. The maximum precision was on the order of 1×10^{-14} when using a time step of 1×10^{-5}. At this point, we reached the maximum precision of this method, and any further value of *dt* would not improve the precision. As we can see, the order of the absolute error lies approximately on the line of x^4.

Dichotomic scheme

Lastly, we tested the final method in which we ran the RK4 algorithm recursively in the region where the function changes sign, halving the time step until we achieved the desired tolerance between points (say, 1×10^{-15}:

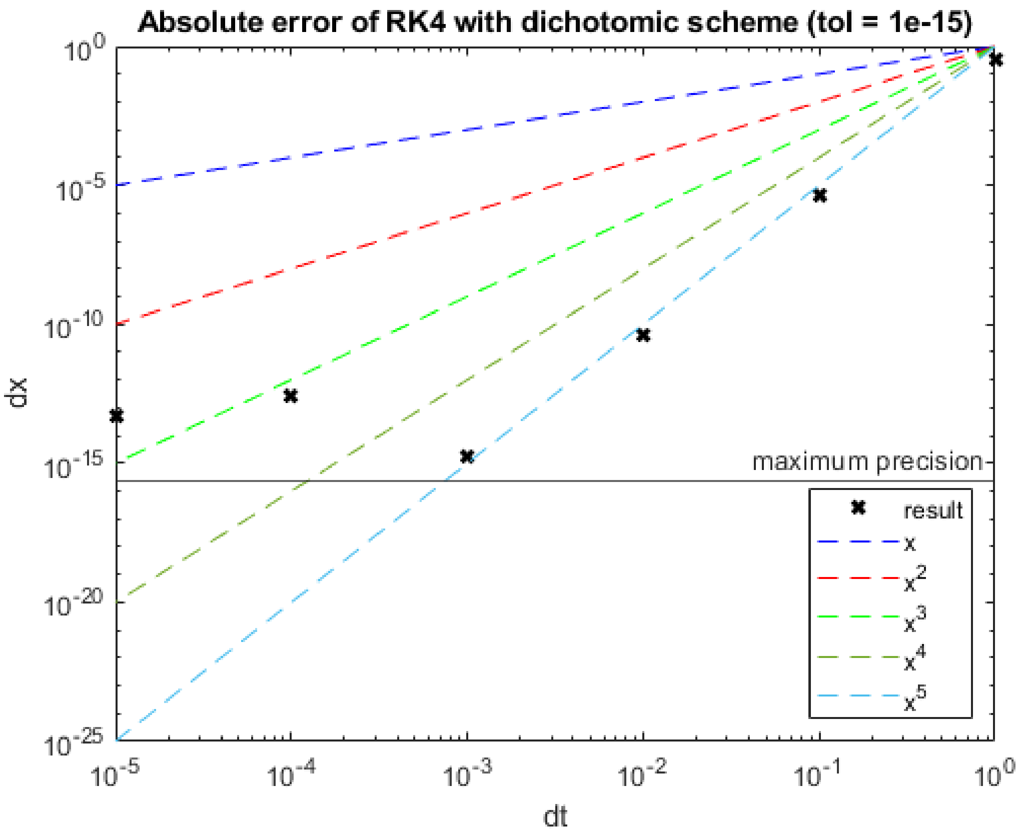


Figure 19: Absolute error of RK4 integration of the harmonic oscillator system combined with with a recursive reduction of the time step to achieve the desired error tolerance of 1×10^{-15} at the final point.

With this method the maximum precision was of the order of 1×10^{-15} when using a time step of only 1×10^{-3}. At this point, we reached the maximum precision of this method, and any further value of dt would not improve the precision. As we can see, the order of the absolute error lies on the line of x^5. This is the best of the methods tested.

Implementation on the example case

When running this test on the example with parameter values of $a_2 = -10$, $b_2 = 2.2$, $c_2 = 0.7$, $\alpha_2 = -71.22$, and $\beta_2 = 0.0015$, the precision if the results was several orders of magnitude lower. Running the test on the cycle that goes through the point $(2.1837, 0)$, we obtain the following results:

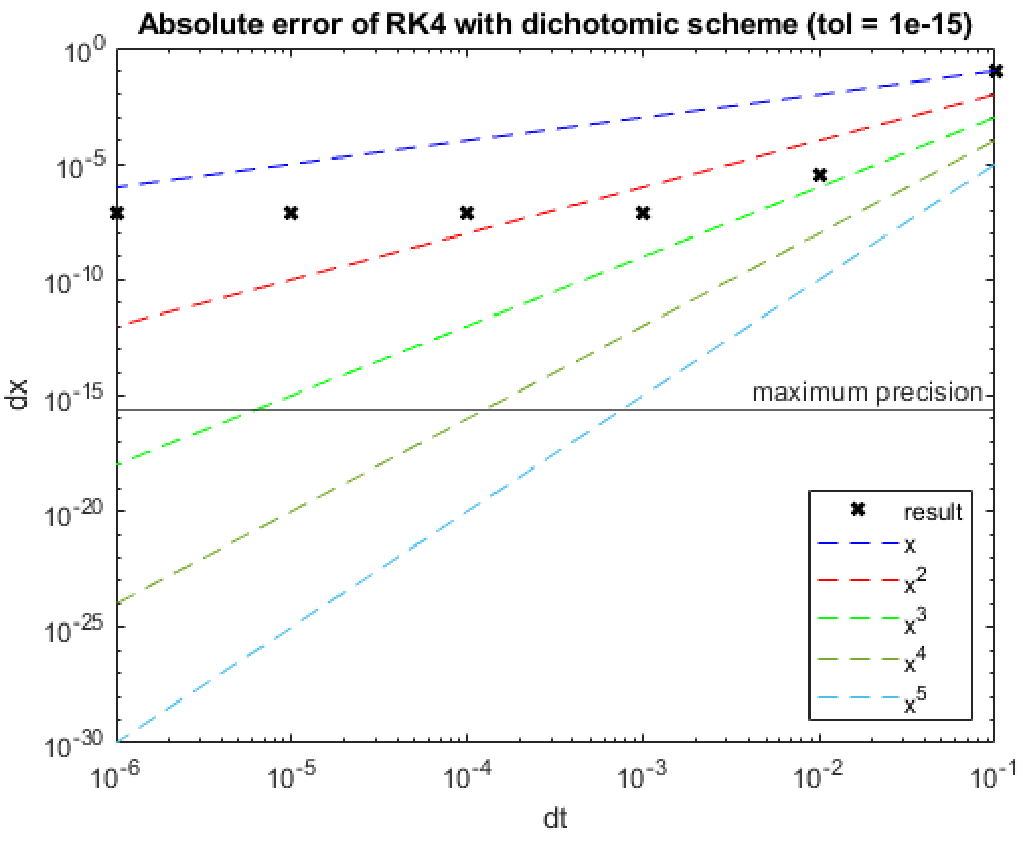


Figure 20: Absolute error of RK4 integration of the limit cycle of the example case combined with with a recursive reduction of the time step to achieve the desired error tolerance of 1×10^{-15} at the final point.

While the precision is worse than that of the harmonic oscillator, an absolute error of 1×10^{-8} while using a time step of 1×10^{-3} is still very accurate for the purposes of this thesis. In this case, the results obtained with inverse quadratic interpolation are very similar.

The reduction in precision most likely arises due to the increase in complexity of the system, and the fact that the point $(2.1837, 0)$ is obtained experimentally, so it is not an exact value, and therefore a source of error capping the maximum precision obtainable.

5.6 Parallel implementation

With the correctness of the code tested, and the precision of the method benchmarked, the final step was to develop a parallel implementation of the code.

5.6.1 Strategy and kernel programming

One of the most common tasks in CUDA programming is to parallelise a loop using a kernel. Common CUDA guidance is to launch one thread per data element, which means that to parallelise the loop we write a kernel that assumes we have enough threads to more than cover the array size. This is sometimes referred to as a monolithic kernel.

However, for problems as complex as what we're dealing with, rather than assume that the thread grid is large enough to cover the entire data array, we instead add a grid-stride loop to our kernel. That is, we write a kernel that loops over the data array one grid-size at a time. Notice that the stride of the loop is

$$blockDim.x * gridDim.x$$

which is the total number of threads in the grid. So if there are 1280 threads in the grid, thread 0 will compute elements 0, 1280, 2560, etc. This is why it is called a grid-stride loop. By using a loop with stride equal to the grid size, we ensure that all addressing within warps is unit-stride, so we get maximum memory coalescing, just as in the monolithic version.

The resulting kernel function something like this:

Algorithm 1 GPU Kernel

```
1: procedure KernelFunction
2:     $index \leftarrow (blockId.x - 1) * blockDim.x + threadId.x$
3:     $stride \leftarrow blockDim.x * gridDim.x$
4:
5:     for $i \leftarrow index : stride :$ length($initialPoints$) do
6:         if $i <$ length($initialPoints$) then
7:             $dx(i) \leftarrow RK4(initialPoints(i), timeStep)$
```

There are several benefits to using a grid-stride loop.

Scalability and thread reuse. By using a loop, you can support any problem size, even if it exceeds the largest grid size your CUDA device supports. Moreover, you can limit the number of blocks you use to tune performance. For example, it's often useful to launch a number of blocks that is a multiple of the number of multiprocessors on the device to balance utilization. When you limit the number of blocks in your grid, threads are reused for multiple computations. Thread reuse amortises thread creation and destruction cost along with any other processing the kernel might do before or after the loop (such as thread-private or shared-data initialisation).

Debugging. By using a loop instead of a monolithic kernel, you can easily switch to serial processing by launching one block with one thread.

Portability and readability.The grid-stride loop code is more like the original sequential loop code than the monolithic kernel code, making it clearer for other users. [24]

5.6.2 Performance

To benchmark the performance, we ran a series of tests on my NVIDIA GeFORCE GTX 1070 GPU, which has a clock frequency of 1594 MHz (around 0.6ns for a single period).

First, we tested our implementation of RK4. We launched the kernel sequentially for some 200.000 trajectories. On average, it took about 651.5 iterations of RK4 to compute each trajectory, each iteration lasting on average 33.49ns, which is reasonable looking at the clock time. If it takes around 21.82μs per trajectory, in a month we should be able to compute 2.628×10^{12} trajectories.

Then, we tested to see how the parallel implementation performs compared to the sequential one. To test for this we called the kernel with different number of points to see how the number of trajectories to compute affects performance. The results are the following:

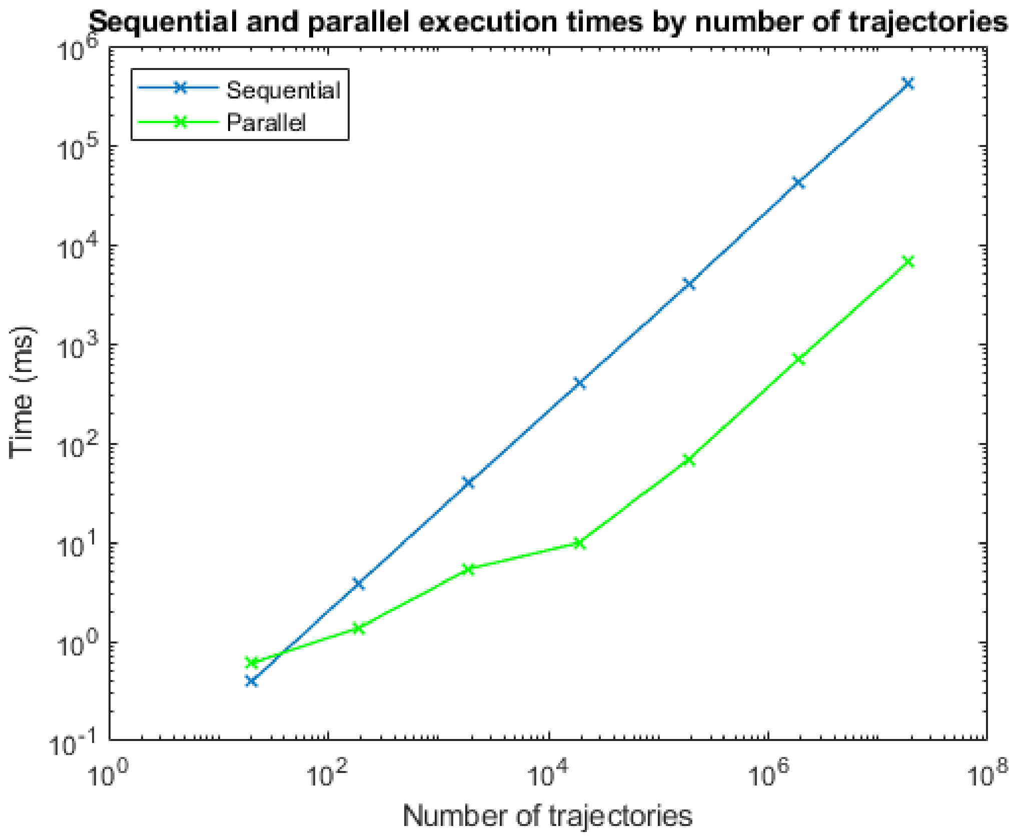


Figure 21: Log-log plot of the time taken to sequentially and concurrently execute the kernel with different number of trajectories. The sequential time (in blue) is directly proportional to the number of trajectories, whereas the parallel execution benefits from larger computations up to a certain extent.

We observed that, for the sequential execution, the execution time is directly proportional to the number of trajectories computed. In fact, when we re-plotted this with the average execution time, we concluded this was a linear relationship, as one might expect:

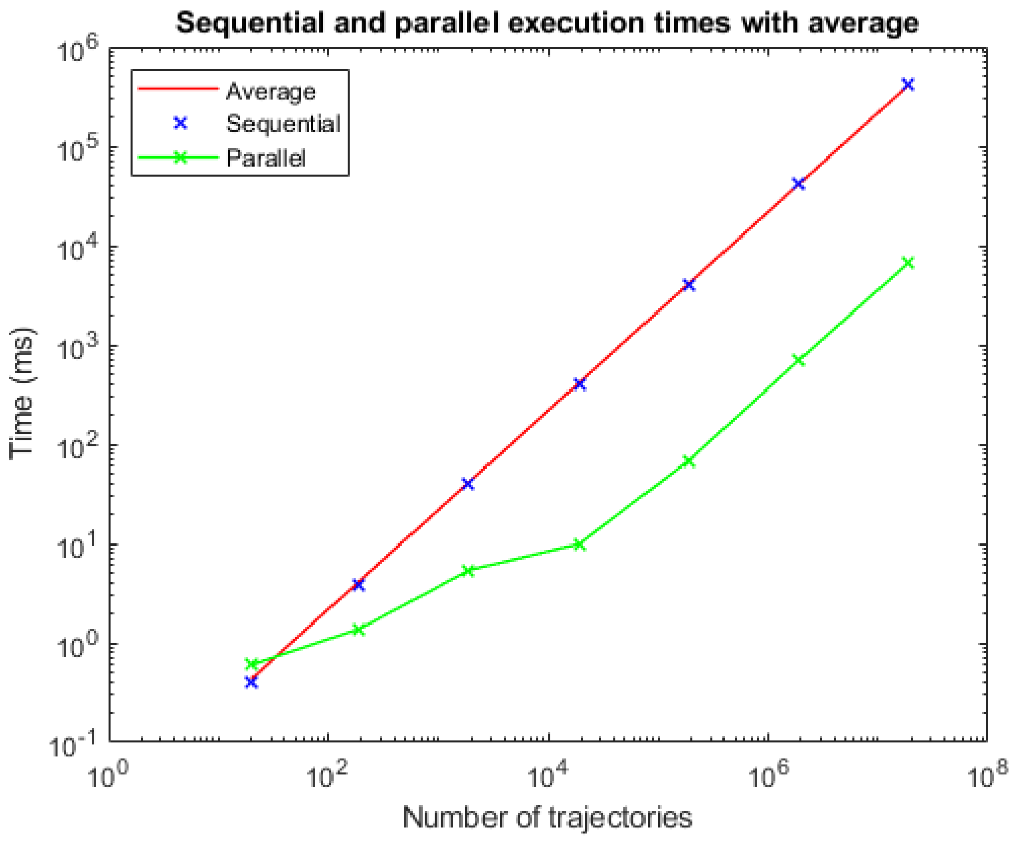


Figure 22: Log-log plot of the time taken to sequentially and concurrently launch the kernel with different number of trajectories. In red is the line corresponding to the average execution time per trajectory multiplied by the number of trajectories computed.. The sequential times are indicated by the blue crosses, and the parallel by the green. We can see that the sequential execution time lines up with the line of the average expected execution time

We also concluded that the execution time is generally much faster when the kernel is executed concurrently. In this graph, we can also observe the effect of the parallelisation overhead for small vectors. The larger the vector, the less significant these overheads become until they are insignificant.

We defined the speed-up of the parallel execution as shown in the next equation:

$$\textit{speed-up} = \textit{sequential execution time} \,/\, \textit{parallel execution time}$$

With the previous execution times, we obtained the following speed-ups:

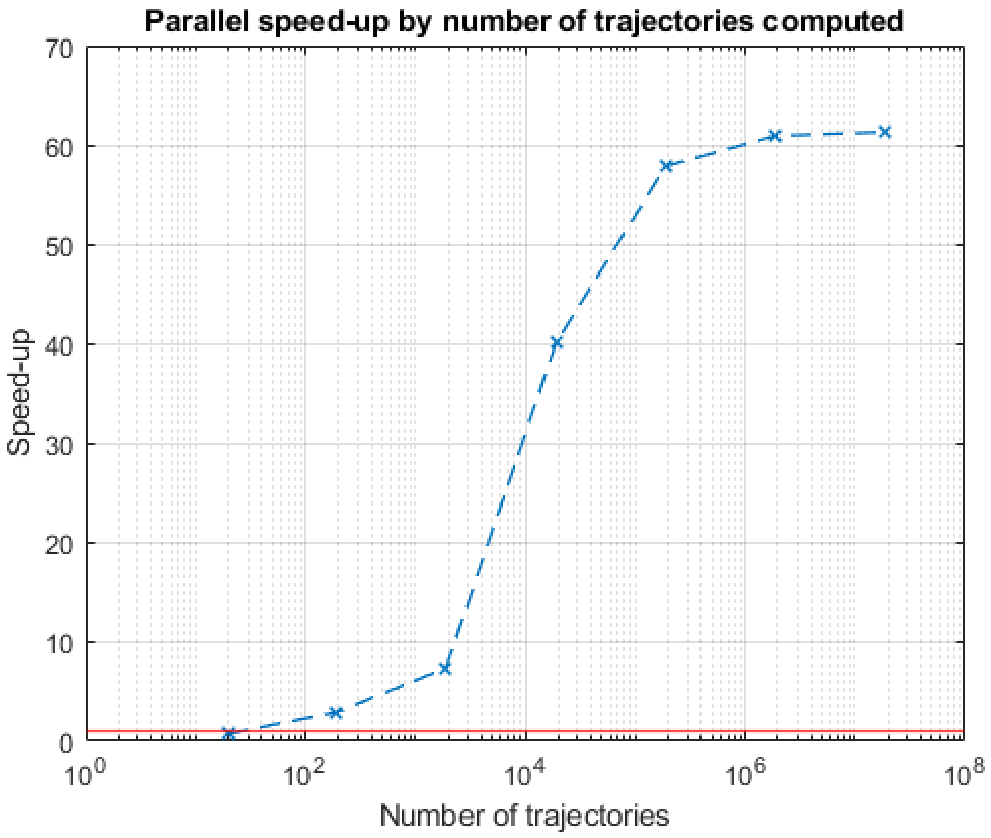


Figure 23: Speed-ups obtained by the parallel kernel for different numbers of trajectories computed. In red is the speed-up of 1. We can see that below 10 trajectories the resulting speed-up is lower than 1, which means that using the parallel implementation is slower than sequentially computing it.

Figure 23 shows how, for lower number of trajectories, the speed-ups achieved were very low. This was most likely due to the fact that the faster execution of the few computations does not make up for the parallelisation overheads. As the number of trajectories computed increased, with the more efficient scheme and combined with the re-utilisation of threads, we got much better speed-ups, seemly plateauing at about 60 for the current parallelisation strategy.

6 Conclusions and future work

6.1 Conclusions

In this thesis we developed a program to integrate systems of two second order polynomial ODEs. We compared different integration methods, and found that an algorithm based on fourth order Runge-Kutta to be best suited to our needs.

We performed extensive testing of convergence with respect to the time step, made a study of the accuracy of as a function of the time step for different integration schemes, and studied the accuracy of different interpolation methods used to find the point where the trajectory crosses the x-axis, comparing it to the machine ϵ. We compared the results obtained with the analytical results for a harmonic oscillator.

We developed our own method for detecting limit cycles based on Poincaré mapping and were able to replicate the results obtained in the literature.

We tested two different parallelisation strategies using MATLAB and Julia, the latter obtaining a speed of up to 60 in the tests run.

6.2 Future work

As possible future work, the developed code can be used on CUDA clusters (such as MinoTauro in Barcelona Supercomputing CXenter (BSC)).

Additionally, we would like to explore further parallelisation strategies and implementations to improve on the speed-up obtained.

References

[1] N. Kuznetsov, O. Kuznetsova, and G. Leonov, “Visualization of four normal size limit cycles in two-dimensional polynomial quadratic system,” *Differential equations and dynamical systems*, vol. 21, no. 1, pp. 29–34, 2013. Accessed: 2022-03-21.

[2] InsideHPC, “What is high performance computing?.” https://bit.ly/3szdUNW, May 2015. Accessed: 2022-03-01.

[3] InfoWorld, “What is cuda? parallel programming for gpus.” https://bit.ly/3Kaf74C, Aug 2018. Accessed: 2022-03-01.

[4] Wikipedia, “Limit cycle.” https://bit.ly/3IyGiWn, Dec 2021. Accessed: 2022-03-01.

[5] X. Sun and J. Lei, *Limit Cycle*, pp. 1126–1127. New York, NY: Springer New York, 2013. https://bit.ly/3MdSMof, Accessed: 2022-03-01.

[6] Wikimedia, “Van der pol stable limit cycle.” https://bit.ly/3vu8DJr. Accessed: 2022-03-01.

[7] Wikipedia, “Hilbert’s problems.” https://bit.ly/36SKNwV, Jan 2022. Accessed: 2022-03-01.

[8] C. Rousseau, “Mathematical developments around hilbert’s 16th problem,” 2007. https://bit.ly/3IAtPBq, Accessed: 2022-03-01.

[9] Kiddle, “Hilbert’s problems facts for kids.” https://bit.ly/3MdAVxJ. Accessed: 2022-03-01.

[10] J. Llibre, “Sobre el problema 16 de hilbert,” *Gaceta de la Real Sociedad Matematica Española*, vol. 18, no. 3, pp. 543–554, 2015. https://bit.ly/3IG4FBy, Accessed: 2022-03-01.

[11] P. Pedregal, “Hilbert’s 16th problem.” https://bit.ly/3psAGFD, 2021. Accessed: 2022-03-01.

[12] Wikipedia, “Hilbert’s sixteenth problem.” https://bit.ly/3K1p3wU, Feb 2022. Accessed: 2022-03-01.

[13] G. Leonov, “Effective methods for investigation of limit cycles in dynamical systems,” *Applied Mathematics and Mechanics*, vol. 74, no. 1, pp. 37–73, 2010.

[14] V. Arnol’d, “Experimental mathematics,” *Fazis*, 2005.

[15] A. Boné Ribó, “High-performance simulation of the 16th hilbert’s problem,” Jun 2021. Accessed: 2022-06-13.

[16] T. Besard, C. Foket, and B. De Sutter, “Effective extensible programming: Unleashing julia on gpus,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2019.

[17] "Power supply calculator - psu calculator." https://outervision.com/power-supply-calculator. Accessed: 2022-03-14.

[18] "Precio del kwh de luz por horas." https://tarifaluzhora.es. Accessed: 2022-03-14.

[19] "Limit cycle poincaré map." https://bit.ly/39Bw5w2. Accessed: 2022-06-18.

[20] "Runge-kutta method — sciencedirect." https://bit.ly/3l2CbHD. Accessed: 2022-05-10.

[21] "Runge–kutta methods." https://bit.ly/3whC8gD, Apr 2022. Accessed: 2022-05-10.

[22] "Numerical differential equations - applied data analysis and tools." https://bit.ly/3PcEt51. Accessed: 2022-05-11.

[23] "ode45 - matlab help center." https://bit.ly/3ypDSad. Accessed: 2022-05-11.

[24] "Cuda pro tip: Write flexible kernels with grid-stride loops." https://bit.ly/3MVhwkh, Oct 2021. Accessed: 2022-06-14.