

Concerning the Stability of Complex Time Steppers

Cian J. Duggan

August 30, 2024

Contents

1	Introduction	1
2	Definitions and Concepts	2
2.1	Numerical Methods	2
2.2	Time Steps	2
2.3	Error	2
2.4	Stability	2
3	The Exponential Decay Problem	3
3.1	A Graphical Representation	3
3.2	The Maclaurin Series	3
3.3	Proponents of the Exponential Decay Problem	4
4	Stability Analysis	5
4.1	Introduction	5
4.2	Numerical Method: Forward Euler	5
4.3	The Stability Function and corresponding Stability Region	5
4.3.1	Stability Region for Euler's Forward Method	6
4.3.2	Stability Region for Euler's Backward Method	6
4.3.3	Stability Region for Runge-Kutta 4	7
4.4	Interpretation of Stability Regions	8
4.4.1	Euler's Forward Method	8
4.4.2	Euler's Backward Method	9
4.4.3	Runge-Kutta 4	9
4.5	Stability Magnitudes	10
4.5.1	Euler's Forward Method	10
4.5.2	Euler's Backward Method	11
4.5.3	Runge-Kutta 4	11
5	Complex Time-Steps	13
5.1	Complex 2-Step	13
5.2	Complex Conjugate Pairs	13
5.3	Stability of 2-Step Methods for Complex Conjugate Step Pairs	13
5.3.1	Euler's Forward 2-Step	14
5.3.2	Euler's Backward 2-Step	15
5.3.3	Runge-Kutta 4 2-Step	16
5.4	Varying b for Complex Conjugate Step Pairs	17
5.4.1	Euler's Forward	17
5.4.2	Euler's Backward	17
5.4.3	Runge-Kutta 4	17
5.5	Varying a	18
5.5.1	Euler's Forward	18
5.5.2	Euler's Backward	18
5.5.3	Runge-Kutta 4	18
6	Conclusions	19
6.1	Summary	19
6.2	Personal Reflection	19
6.3	Future Work	19
6.4	Acknowledgements	19
Appendices		22
A	Additional Explorations	22
A.1	Numerical Method: 2-step Abysmal Kramer-Butler Method	22
A.2	Runge-Kutta 4 Complex Step Pairs Conjecture	22
B	Code	22
B.1	Exact Solution for Exponential Decay	22
B.2	Basic Stability Regions	25
B.3	Exact vs Methods	27
B.4	Stability Magnitude 3D	30
B.5	Stability Magnitude Contour	32
B.6	Stability Magnitude Solutions	34

B.7	Runge-Kutta 4 Stability Magnitude	36
B.8	Real VS Complex Stability Regions	37
B.9	Video Frame Generation	39
B.10	Frame Stitching for Video	41

1 Introduction

A paper by Lloyd N. Trefethen [1] offers two definitions of numerical analysis:

The study of rounding errors.

The study of algorithms for the problems of continuous mathematics.

Trefethen argues that the first definition, though an accurate summation of the field's history, does not serve to entice the curious to explore the field. He proposes the second as a more compelling definition, one that emphasises the field's role in solving real-world problems, and inspires curiosity.

He offers an optimistic view of the field, one that is not bogged down by the minutiae of rounding errors, but rather one that is focused on the algorithms that make numerical analysis possible. More to ground, the majority of numerical analysis is concerned with the speed of convergence and minimization of error.

Through my work on this paper, I have honed my own definition:

Numerical analysis is the field which attempts to straddle the gaps between the discrete and the continuous.

In this paper, we will take a look at the stability of various numerical methods when applied to problems of differential calculus.

In particular, we wish to:

- Introduce key terminology and concepts in the field of numerical analysis.
- Establish the Exponential Decay Problem as the toy problem for analysis.
- Define stability, and its associated concepts, in the context of numerical analysis.
- Establish the concept of complex time steps for numerical methods.
- Develop a framework for the analysis of the stability of numerical methods.
- Apply this framework to particular numerical methods.
- Expand upon the results of this to contrast the stability regions for real and complex time steps.

2 Definitions and Concepts

2.1 Numerical Methods

Numerical methods are techniques used to approximate solutions to problems that cannot be solved exactly.

Numerical methods are prevalent when working with problems of differential calculus in a computational context, where many problems do not have closed-form solutions.

A numerical method can be interpreted as an algorithm; a series of steps that can be followed to approximate a solution. Many numerical methods exist, each with their own properties and trade-offs.

In this respect, different methods may be more or less suitable for different problems.

Core to understanding which to use for a particular problem are the concepts of *error* and *stability*.

Before discussing these properties, however, we must understand how a numerical method works.

The basis of many numerical methods is the concept of a *time step*.

2.2 Time Steps

Let's take a step back to when we first studied derivatives.

Newton's Difference Quotient should be familiar: $f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h)-f(x)}{h}$

Viewing Newton's Difference Quotient in the context of numerical methods, we call h a *time step*.

Due to computational constraints, we cannot take h to be infinitesimal. (A computer's memory is as small as it is big) Instead, we take h to be a small, finite number.

This defines the concept of a *step size*.

For numerical methods, we use $y(t)$ to denote the solution to a differential equation at time t .

We approximate $y'(t)$ by $\frac{y(t+h)-y(t)}{h}$ using Newton's Difference Quotient.

Note this is an approximation because we have dropped the limit; h is not infinitesimal.

This gives $y(t+h) \approx y(t) + hy'(t)$; a first-order approximation of y a small time step h in the future.

For a numerical method we say $y(t+h) = y(t) + hy'(t)$

2.3 Error

In numerical analysis, *error* is the difference between the numerical solution and the exact solution to a problem.

Error can be caused by many factors, such as the choice of numerical method, the choice of time step, or the precision of the computer.

Error can be classified into two categories: truncation error and rounding error.

Truncation Error is the error introduced by approximating a problem, such as using a finite time step.

Rounding Error is the error introduced by the finite precision of a computer.

Error is a crucial concept in numerical analysis, as it determines the accuracy of a numerical method.

Any error mentioned in this document refers to the truncation error; we won't be looking at how computers run these calculations and the rounding errors that come with it.

2.4 Stability

In numerical analysis, a method is said to be *stable* if small deviations in the input do not lead to large perturbations in the output. In the context of differential equations, a method is said to be stable if the solution does not grow to be unbounded as the number of time steps increases. (This, of course, only applies if the exact solution is bounded. We will restrict ourselves to this with our analysis.)

When solving differential equations numerically, the choice of time step is crucial. If the step size is too large, the solution may become unstable; the numerical solution will diverge from the analytic solution in proportion with the number of steps. If the step size is too small, the solution may be accurate and stable, but the computation may be too slow.

The tradeoff between error, stability and compute is a common theme in numerical analysis.

The sweet spot for maximal efficiency depends entirely on the choice of numerical method.

This marks the core motivation for this report; solidifying an understanding to better inform the choice of method and step size for a given problem.

3 The Exponential Decay Problem

We would like a simple toy problem with which we can build a framework for the analysis of the stability of a numerical method.

We turn to a classic problem in numerical analysis: the Exponential Decay Problem.

Consider a quantity y that decays relative to its current value.

The quantity at time t is $y(t)$ and the rate of decay is λ .

For the solution to actually decay, we require $\operatorname{Re}(\lambda) < 0$.

The Exponential Decay Problem is given by the ODE

$$y'(t) = \lambda y(t) \quad \text{with} \quad y(0) = y_0$$

This has the exact solution $y(t) = y_0 e^{\lambda t}$, hence the name 'Exponential Decay'.

For the sake of keeping things neat, we will take $y_0 = 1$ for the rest of this paper.

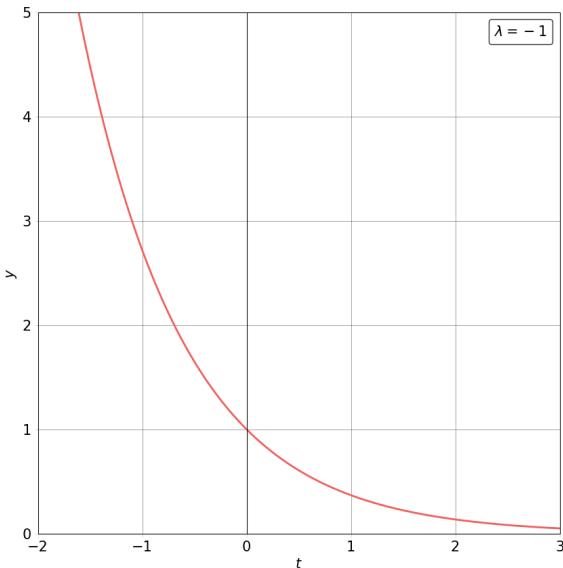
This poses no lack of generality, as we can always scale the solution by any initial value, both exact and numerical.

The exact solution we will consider is

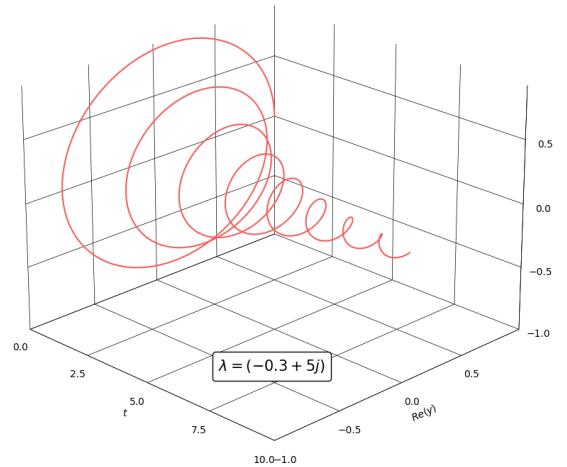
$$y(t) = e^{\lambda t}$$

3.1 A Graphical Representation

Appendix B.1 contains the Python for these graphs.



If we restrict λ to \mathbb{R}^- , we have a simple exponential curve that decays to zero as $t \rightarrow \infty$.



If we allow $\lambda \in \mathbb{C}$ and $\operatorname{Re}(\lambda) < 0$, we get a spiral in the complex plane, the radius of which decays to zero as $t \rightarrow \infty$.

3.2 The Maclaurin Series

The Taylor series is given by:

$$y(t) = \sum_{n=0}^{\infty} \frac{y^{(n)}(a)}{n!} (t-a)^n$$

Setting $a = 0$, we get the Maclaurin series for the exact solution:

$$y(t) = \sum_{n=0}^{\infty} \frac{(\lambda)^n e^0}{n!} t^n = \sum_{n=0}^{\infty} \frac{(\lambda t)^n}{n!} \quad \text{which we will denote } M(y)$$

This is a useful form for the exact solution – we will see it pop up again later.

3.3 Proponents of the Exponential Decay Problem

There are a few pros to this choice of problem:

- The exact solution is known and easily computed; it can be used to evaluate a numerical solution.
- The solution is a straightforward and well understood function.
This simplicity allows us to focus on the numerical method.
- The problem is linear in $y(t)$, making the problem simple to work with.
Again, our focus can stay on the numerical method.
- The graph of the exact solution is easy to visualise; the more chaotic outputs of the numerical methods can be compared to the smooth curve of the exact solution easily.
- Exponential decay is a model for many physical processes, including radioactive decay, population decline, and capacitor discharge.
Quantities that decay relative to their current value appear frequently in nature.
This means readers from many different disciplines may already possess an intuition for the problem.
- The problem is simple to generalise to complex numbers, allowing us to explore the stability of numerical methods in the complex plane.
- The Exponential Decay Problem is *stiff*; for certain method-step pairs, the numerical solution may oscillate wildly.
This is exactly the kind of behaviour we want to avoid in a numerical method, when we are looking for stability.

Normally, when using the Exponential Decay Problem in a numerical analysis context, we would allow $\lambda \in \mathbb{C}$ with $Re(\lambda) < 0$.

However, for the purposes of this paper, we will restrict λ to \mathbb{R} as we will be looking at $h \in \mathbb{C}$ and we don't want to overcomplicate things.

Unless otherwise mentioned, we will assume that $\lambda \in \mathbb{R}^-$ for the rest of this paper.

We will use the Exponential Decay Problem throughout this paper to derive a useful framework for the analysis of the stability of a numerical method.

4 Stability Analysis

4.1 Introduction

In this section we will lay out a framework for analysing the stability of a numerical method.

To begin, we will introduce the concept of the stability function.

We will show how it can be used to define a stability region.

We will explore the stability region as the set of values for which a numerical method is stable.

Finally, we will extend this framework to view the stability of a numerical method in the context of complex time steps.

4.2 Numerical Method: Forward Euler

Let's consider a simple numerical method applied to the Exponential Decay Problem.

The Forward Euler method is a first-order numerical method for solving ODEs with a given initial condition.

Its algorithm can be defined as:

$$y(t_{j+1}) = y(t_j) + hy'(t_j)$$

Where h is the time step and y_j is the numerical solution at time t_j , j steps on from $t_0 = 0$.

This means $t_j = hj$.

For the Exponential Decay Problem, the Forward Euler method can be written as:

$$y_{j+1} = y_j + h\lambda y_j \quad \text{where } y_0 = 1$$

This gives us the following algorithm:

$$y_{j+1} = (1 + h\lambda)y_j$$

This gives an approximation of the exact solution at time t_j as follows:

$$y_j = (1 + h\lambda)^j y_0 = (1 + h\lambda)^j = y(t_j) = y(hj) \approx e^{\lambda hj}$$

We can see that the Forward Euler method is stable if $|1 + h\lambda| < 1$;

both the exact solution and our approximation will decay to zero as $t \rightarrow \infty$.

The stability is dependent on the time step h and the value of λ .

We can write this as $s(\lambda, h) = 1 + h\lambda$.

By analysing s for different values of λ and h ,

we can infer the stability of the Forward Euler method for the Exponential Decay Problem.

We call s the *stability function* of the Forward Euler method.

4.3 The Stability Function and corresponding Stability Region

By the same methodology, we can define the stability function for any numerical method by writing the algorithm in the form $y_{j+1} = s(\lambda, h)y_j$.

$s(\lambda, h)$ is the *stability function* of the numerical method with a time step h .

Note: This is equivalent to $y_j = s(\lambda, h)^j y_0$

The *stability region* of a numerical method is the set $S = \left\{ (\lambda, h) \mid |s(\lambda, h)| < 1 \right\} \subset \mathbb{C}$

This follows from the definition of stability for our Exponential Decay Problem;

a method is stable if the numerical solution decays to zero as $t \rightarrow \infty$.

Clearly, $\lim_{j \rightarrow \infty} y_j = \lim_{j \rightarrow \infty} s(\lambda, h)^j y_0 = 0 \iff |s(\lambda, h)| < 1$. Now, let's explore some examples of stability regions for different numerical methods.

4.3.1 Stability Region for Euler's Forward Method

Euler's Forward Method has the stability function

$$s(\lambda, h) = 1 + \lambda h = \sum_{n=0}^1 \frac{(\lambda h)^n}{n!} = M(y) + \mathcal{O}((\lambda h)^2)$$

The corresponding stability region is

$$S = \{(\lambda, h) \mid |1 + \lambda h| < 1\}$$

We can see this region plotted in red on the right; an open unit circle centred at -1 .

For a given $\lambda \in \mathbb{C}$, the method is stable for any step-size $h \in \mathbb{R}$ such that $|1 + \lambda h| < 1$.

Expanding $\lambda = a + bi$, we get the restriction

$$\begin{aligned} |1 + (a + bi)h| &< 1 \implies |(1 + ah) + (bh)i| < 1 \\ \implies a^2h^2 + 2ah + b^2h^2 &< 0 \\ a, b, h \in \mathbb{R} \implies (a^2 + b^2)h^2 &> 0 \\ \implies 2ah &< 0 \text{ and } ||2ah|| > (a^2 + b^2)h^2 \end{aligned}$$

This aligns with our intuition for the problem;

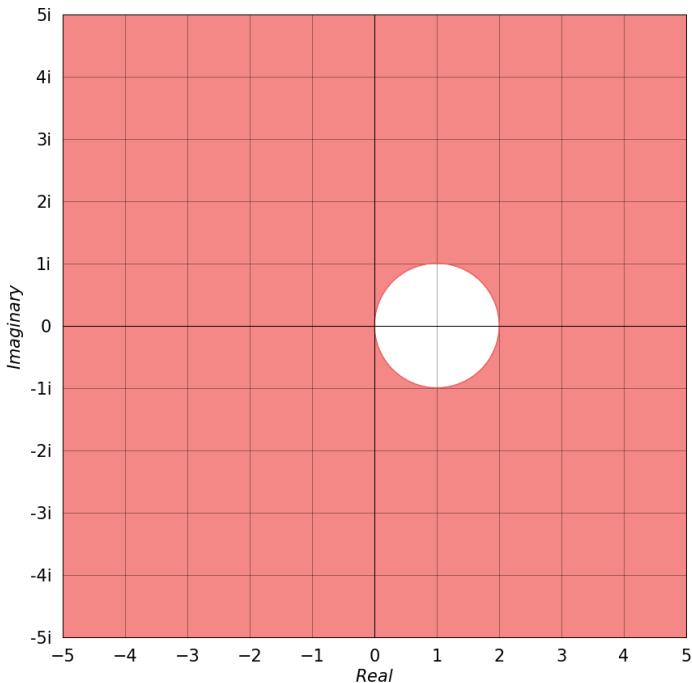
- For the exponential to decay, we must have $Re(\lambda) = a < 0$.
- h must be positive, as it is a time step.

Moreover,

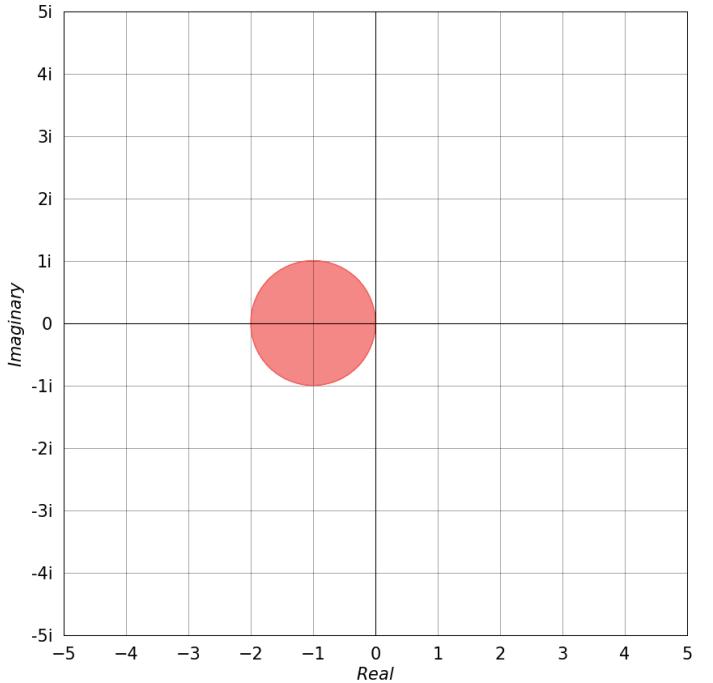
$$0 < (a^2 + b^2)h^2 < ||2ah|| \implies 0 < h < \frac{||2a||}{a^2 + b^2} \implies 0 < h < \frac{2||Re(\lambda)||}{|\lambda|^2}$$

This is also intuitive; if we imagine λ growing linearly, h must shrink quadratically so that their product λh stays inside the stability region.

4.3.2 Stability Region for Euler's Backward Method



See Appendix B.2 for the graphing code.



See Appendix B.2 for the graphing code.

Euler's Backward Method can be written as

$$y_{j+1} = y_j + h \cdot y'(t_{j+1})$$

$$\implies y_{j+1} = y_j + h(\lambda y_{j+1}) \implies y_{j+1} = \frac{1}{1 - \lambda h} y_j$$

Thus, the stability function is

$$s(\lambda, h) = \frac{1}{1 - \lambda h}$$

The corresponding stability region is

$$S = \left\{ (\lambda, h) \mid \left| \frac{1}{1 - \lambda h} \right| < 1 \right\}$$

This is plotted in red on the left; the region outside a unit circle centred at 1 .

The white region of instability is exactly the stability region for Euler's Forward Method, flipped about Imaginary axis.

We could run through the same algebraic procedure as before, expanding λ and rearranging, and we would find that

$$h > \frac{2Re(\lambda)}{|\lambda|^2}$$

This is a more lax restriction than the $0 < h$ that we have already established.

In fact, this tells us that any positive h will give a stable solution, regardless of the value of λ .

This is a property called **Absolute Stability** or **A-Stability**.

This is often characterised by a stability region that covers the entire left half-plane, as we see here.

4.3.3 Stability Region for Runge-Kutta 4

Runge-Kutta 4 can be written as

$$y_{j+1} = y_j + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Where $\phi(t, y) = y'(t) = \lambda y$

$$k_1 = \phi(t_j, y_j) \quad k_2 = \phi\left(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_1\right)$$

$$k_3 = \phi\left(t_j + \frac{h}{2}, y_j + \frac{h}{2}k_2\right) \quad k_4 = \phi(t_j + h, y_j + hk_3)$$

For the Exponential Decay Problem, we get

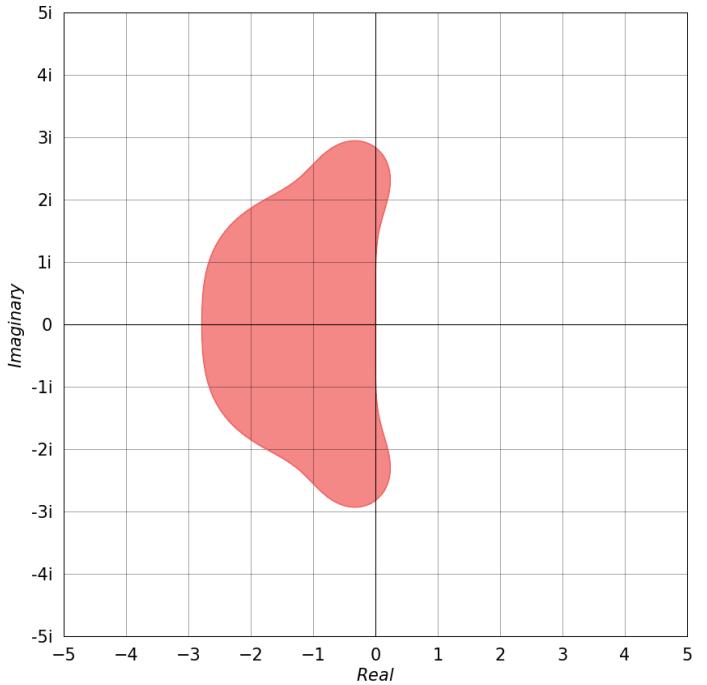
$$y_{j+1} = \left(1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \frac{(\lambda h)^4}{24}\right)y_j$$

The stability function is

$$s(\lambda, h) = \sum_{n=0}^4 \frac{(\lambda h)^n}{n!} = M(y) + \mathcal{O}((\lambda h)^5)$$

The corresponding stability region is

$$S = \left\{ (\lambda, h) \mid \left| \sum_{n=0}^4 \frac{(\lambda h)^n}{n!} \right| < 1 \right\}$$



See Appendix B.2 for the graphing code.

4.4 Interpretation of Stability Regions

The stability region $S = \{(\lambda, h) \mid |s(\lambda, h)| < 1\} \in \mathbb{C}$ is a set of values for which a numerical method is stable for the Exponential Decay Problem.

We always restrict $h \in \mathbb{R}^+$, as it is a time step.

We have two separate cases for λ :

- $\lambda \in \mathbb{R}^-$: S corresponds to $\{s(\lambda, h)^2 < 1\}$. Of course, $\lambda, h \in \mathbb{R} \implies S \subset \mathbb{R}$.

- $\lambda \in \mathbb{C} \setminus \mathbb{R}$ with $\operatorname{Re}(\lambda) < 1$: S corresponds to $\{(s(\lambda, h))(\overline{s(\lambda, h)}) < 1\}$.

From here, one can derive bounds on h for a given value of λ , or vice versa.

We did this with Euler's Forward and Backward methods above.

The Runge-Kutta 4 method is more complicated, as the two cases of λ each give a polynomial of degree 8.

Below we have graphs for the cases where λ is real or complex, the same as we introduced for the Exponential Decay Problem.

In black is the exact solution. In colour are the numerical solutions due to the method in question.

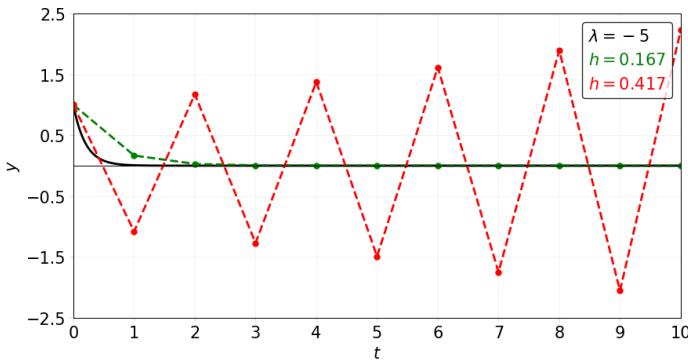
These (λ, h) pairs were found by first picking a value of λ and then finding h values which met the stability condition.

In green $(\lambda, h) \in S$.

In red $(\lambda, h) \notin S$.

Appendix B.3 has the code for all graphs in this section.

4.4.1 Euler's Forward Method



On the left are the aforementioned graphs for Euler's Forward Method.

First is the case where $\lambda \in \mathbb{R}^-$.

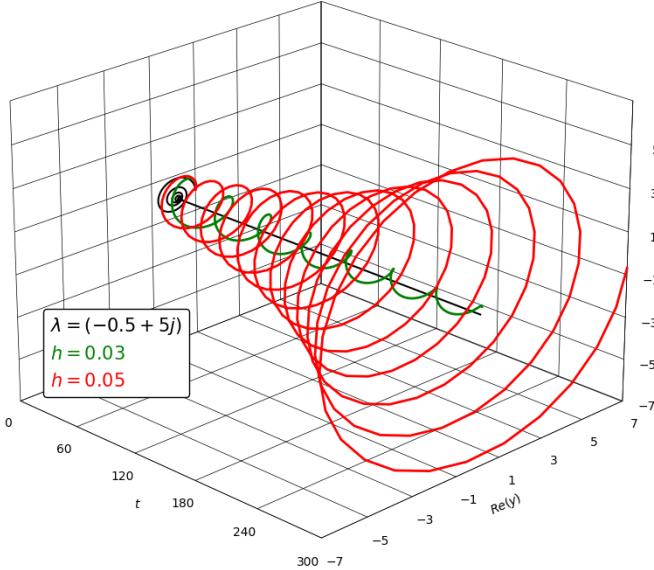
The exact solution decays quickly in black, very close to $t = 0$. The numerical solution converges to the exact solution for $h = 1/6$ in green.

For $h = 5/12$, the numerical solution oscillates and diverges in red.

The oscillation is due to the stability function $s(\lambda, h) = 1 + \lambda h = 1 + -5 \cdot \frac{5}{12} = \frac{-13}{12}$

The j^{th} term of the numerical solution is $s(\lambda, h)^j = \left(\frac{-13}{12}\right)^j$. Of course, natural powers of negative numbers oscillate between positive and negative values.

$\left|\frac{-13}{12}\right| > 1 \implies$ the numerical solution diverges.



Second is the case where $\lambda \in \mathbb{C}$.

The exact solution decays quickly in black, very close to the $t = 0$ plane.

The green, valid h value converges as t increases, but the red, invalid h value diverges.

While this 3D graph might not be the clearest, it took quite a lot of messing with values of h and λ to find one so demonstrative.

The reader is encouraged to play with values for h and λ in Appendix B.3 to see for themselves.

This has the added benefit of allowing you to rotate the graph to see the behaviour from different angles.

4.4.2 Euler's Backward Method

As mentioned before, Euler's Backward Method is A-Stable. This means that for any $\lambda \in \mathbb{C}$ with $\operatorname{Re}(\lambda) < 1$, the method is stable for any $h \in \mathbb{R}^+$.

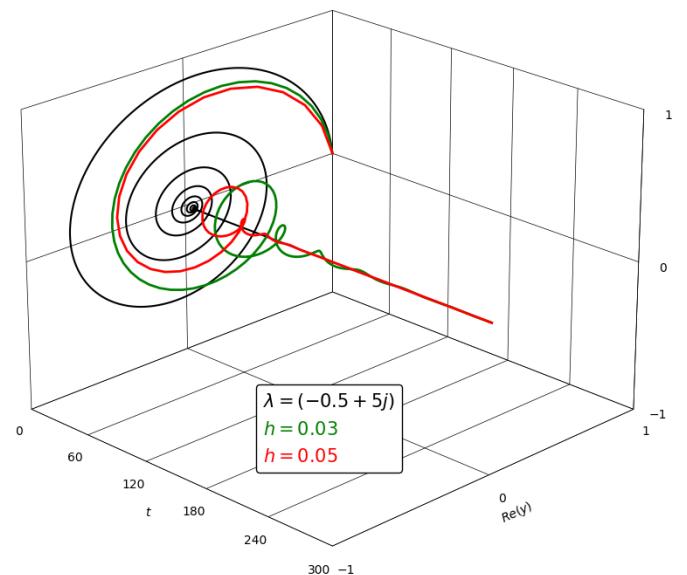
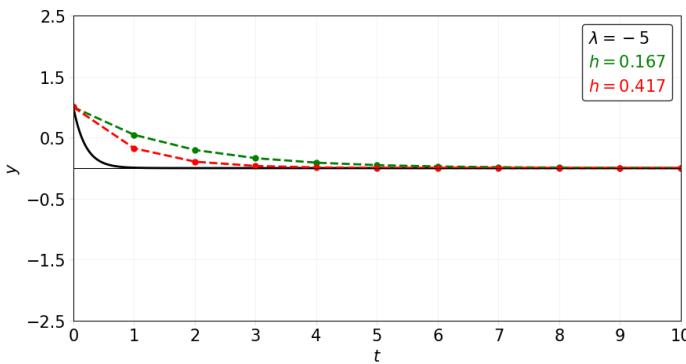
As a result, there is no choice of (λ, h) that will give us a divergent graph.

For λh to land in the white disk of instability, either h must be negative or $\operatorname{Re}(\lambda) > 1$.

Both of these are nonsensical for our Exponential Decay Problem.

Instead, these graphs show two different stable h values for the same λ value.

For $\lambda \in \mathbb{R}^-$ below, the numerical solution converges to the exact solution for both step sizes.



For $\lambda \in \mathbb{C}$ above, the method converges to the exact solution for both step sizes.

There is no divergence, as we have already established. Interestingly, the method converges faster for the larger step size.

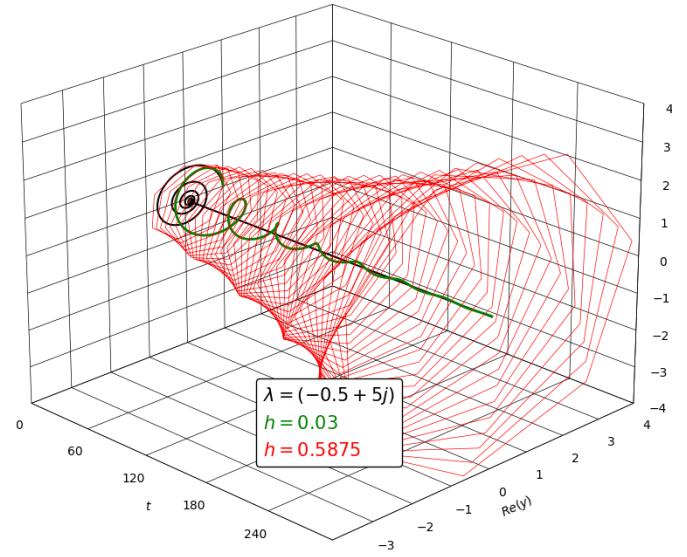
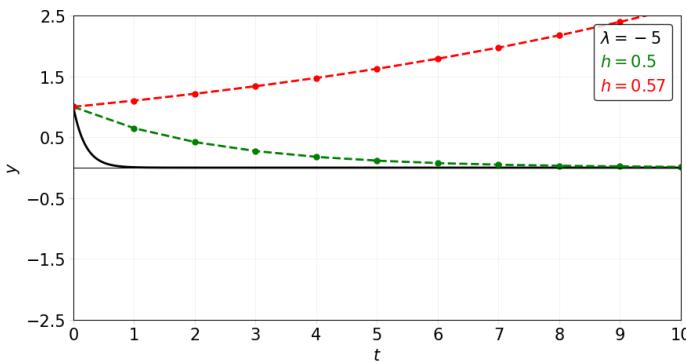
The subsequent section on Stability Magnitudes will explain this behaviour.

4.4.3 Runge-Kutta 4

The same properties hold true for Runge-Kutta 4, as we demonstrated for Euler's Forward, though we choose different values of h and λ for aesthetic clarity.

It was particularly to find a set of values that demonstrated both convergence and divergence, while keeping the exact solution visible.

The linewidth of the unstable h method was decreased to keep everything clear.



4.5 Stability Magnitudes

So far we have been looking at stability regions given by $|s(\lambda, h)| < 1$.

While this bound is a sufficient condition for stability, we are not truly utilising the metric.

In this section, we'll explore $|s(\lambda, h)|$ in a little more detail.

The plots below show the magnitude of the stability function for different values of λh .

We have limited these to display only the magnitudes within the region S for clarity; $|s(\lambda, h)| < 1$.

The 3D plots show the magnitudes $|s(\lambda, h)|$ above the base plane S .

The magnitudes have been projected down onto the base plane as colour plots.

We show these colour plots separately in the second graphs.

While on the first graphs, these colour gradients are continuous, on the second graphs we have discretised them.

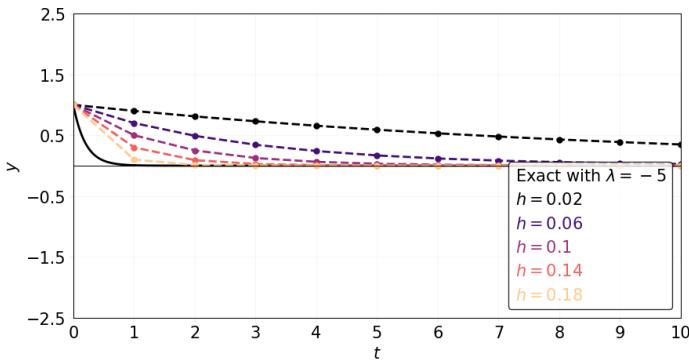
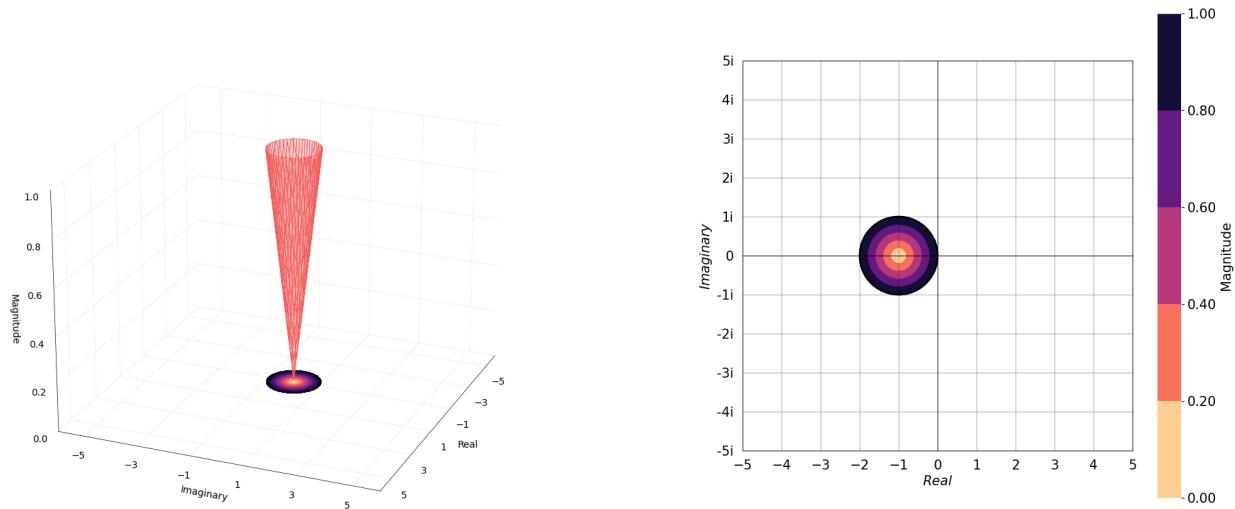
This is so that for the third graphs we can create a correspondance with the plotted numerical solutions.

We'll only present the Exponential Decay graphs for real λ here, as the complex λ graphs are quite cluttered.

Their behavior is the same as for the real λ graphs, with tighter, faster decaying spirals as λh lands in brighter areas of the colour plot.

See Appendices B.4, B.5 and B.6 for the python used to generate graphs 1, 2 and 3 respectively.

4.5.1 Euler's Forward Method



We can see smaller magnitudes correspond to a tighter fit to the exact solution.

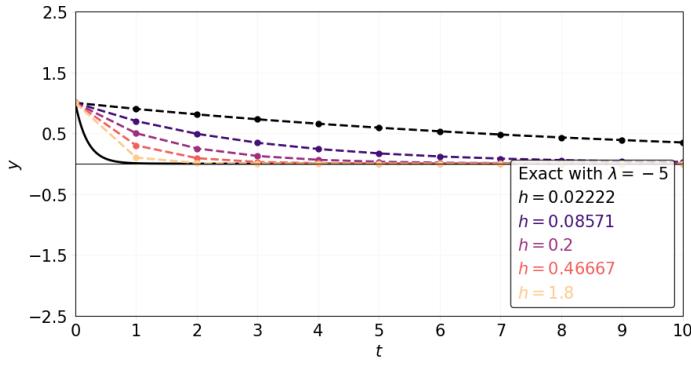
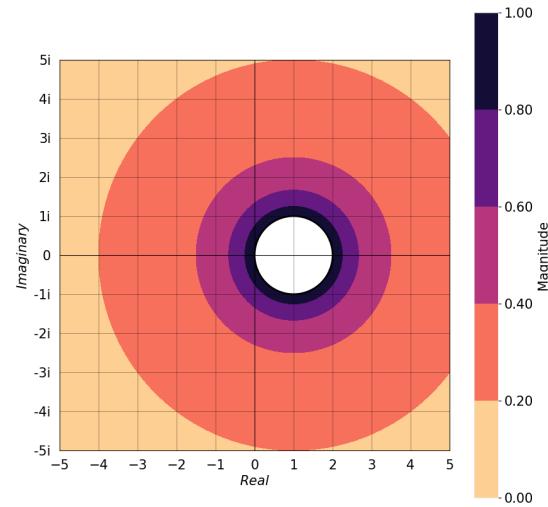
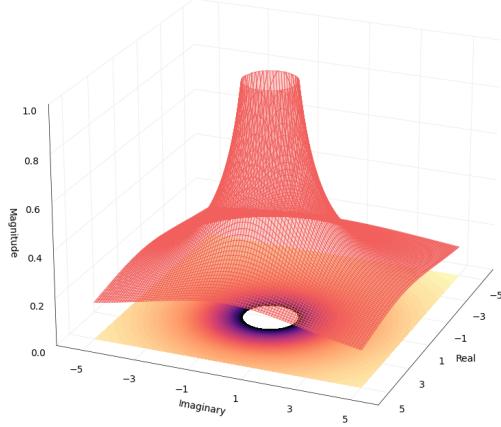
This aligns with our intuition; for a derivative, we allow $h \rightarrow 0$ to obtain the exact solution.

Recall, in a computational context, we're looking for the largest h that will give us a stable solution, that has an error within our tolerance.

Between each point on the graph, $\frac{1}{h}$ calculations are done to find the next point.

For larger h , the method is less accurate, but faster.

4.5.2 Euler's Backward Method



As we have already established, Euler's Backward Method is A-Stable.

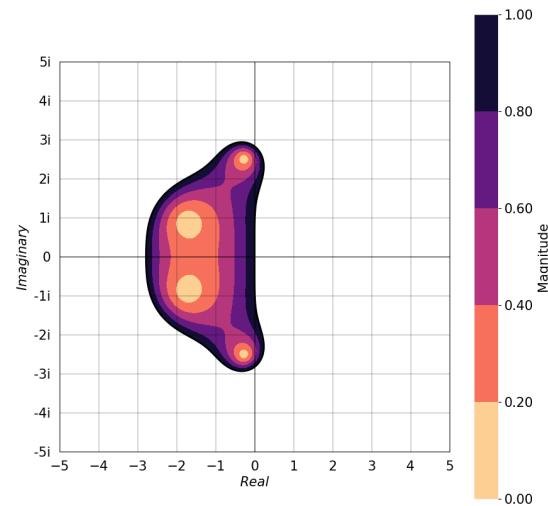
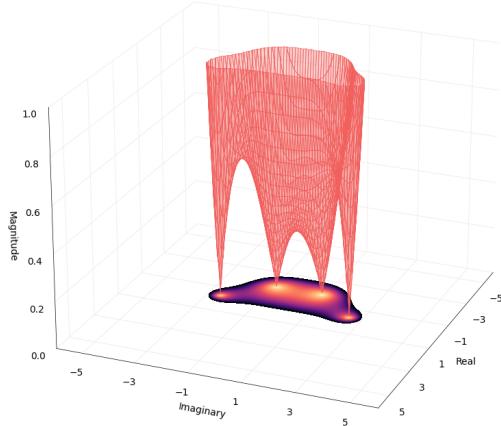
Any choice of $h \in \mathbb{R}^+$ will give a stable solution for any $\lambda \in \mathbb{C}$ with $\operatorname{Re}(\lambda) < 1$.

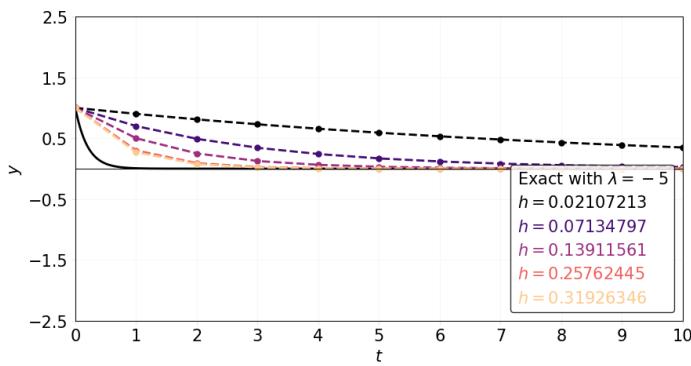
It is impossible to pick such (λ, h) values that will give λh in the white circle.

By nature of the stability function $s(\lambda, h) = \frac{1}{1-\lambda h}$, larger values of h will give a smaller magnitudes; a tighter fit to the exact solution.

This indicates why implicit methods like Euler's Backward are often used for stiff problems; they can be more accurate for larger time steps.

4.5.3 Runge-Kutta 4





The same general observations as Euler's Forward Method hold true for Runge-Kutta 4.

The values of h here were calculated numerically, rather than algebraically.

The code for calculating these values can be found at Appendix B.7.

5 Complex Time-Steps

So far, we have restricted our time-steps to the real domain.

It has often proven useful to extend the analysis of a problem to the complex domain.

Motivated by a paper by *George, Yung and Mangan*[2], we will have a look at the stability of numerical methods when the chosen time-step is complex.

They state:

Most numerical methods for time integration use real time steps. Complex time steps provide an additional degree of freedom, as we can select the magnitude of the step in both the real and imaginary directions. By time stepping along specific paths in the complex plane, integrators can gain higher orders of accuracy or achieve expanded stability regions.

5.1 Complex 2-Step

We want to take an overall step of size $h \in \mathbb{R}$ comprised of two steps $z_1, z_2 \in \mathbb{C}$ such that $z_1 + z_2 = h$.

We can set $z_1 = a + bi$ and $z_2 = h - a - bi$.

For analysing our numerical methods, we require a comparable implementation of the real step setup.

A real 2-step method where we take a step of size $\frac{h}{2}$ allows us to take two steps and go the same distance as the complex 2-step method.

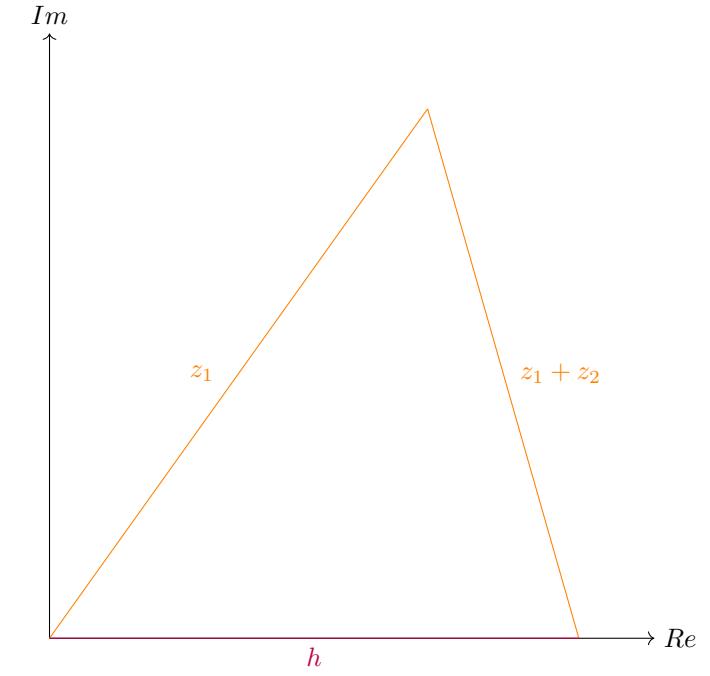
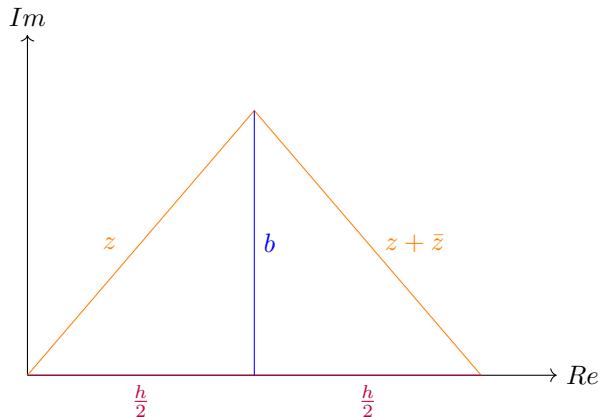
We can write this real 2-step method as

$$y_{j+1} = s(\lambda, \frac{h}{2})y_{j+\frac{1}{2}} = s(\lambda, \frac{h}{2})s(\lambda, \frac{h}{2})y_j = s(\lambda, \frac{h}{2})^2 y_j$$

We can write the complex 2-step method as

$$y_{j+1} = s(\lambda, z_1)y_{j+\frac{1}{2}} = s(\lambda, z_1)s(\lambda, z_2)y_j$$

5.2 Complex Conjugate Pairs



When we restrict the exponential decay problem by $\lambda \in \mathbb{R}$, we also find that $s(\lambda, z_1)s(\lambda, z_2) \in \mathbb{R}$. This is because any value attained by $e^{\lambda t}$ is real. This restricts the values of z_1 and z_2 , as any imaginary terms in $s(\lambda, z_1)s(\lambda, z_2)$ must cancel out.

One particularly simple family of such complex pairs is defined by setting $a = \frac{h}{2}$. This necessitates $z_2 = \bar{z}_1$. We will refer to this family as the *complex conjugate step pairs*.

5.3 Stability of 2-Step Methods for Complex Conjugate Step Pairs

In this section we will compare the stability regions for the real and complex 2-step methods.

The real step size is given by $\frac{h}{2}$.

The complex conjugate step pair is given by $z = \frac{h}{2} + bi$ and $\bar{z} = \frac{h}{2} - bi$.

The plots in this section show the regions of stability $S = \{\lambda h : |s(\lambda, h)| \leq 1\}$ for the case where $b = a = \frac{\lambda h}{2}$.

We will have a look at varying b in a subsequent section.

5.3.1 Euler's Forward 2-Step

Theorem 5.1. *Euler's Forward Complex Step Pairs must be Conjugate for $\lambda \in \mathbb{R}$*

For the exponential decay problem, $y = e^{\lambda t}$, let $\lambda \in \mathbb{R}$

Let $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$ be a complex step pair for Euler's Forward 2-Step Method.
Then we must have $z_2 = \bar{z}_1$

Proof:

$$\lambda \in \mathbb{R} \implies y_j, y_{j+1} \in \mathbb{R}$$

Euler's Forward 2-Step Method is given by:

$$\begin{aligned} y_{j+1} &= s(\lambda, z_1)s(\lambda, z_2) y_j \\ &= (1 + z_1)(1 + z_2) y_j \\ &= (1 + a_1 + b_1 i)(1 + a_2 + b_2 i) y_j \\ &= \left((1 + a_1 + a_2 + a_1 a_2 - b_1 b_2) + (b_1 + b_2 + a_1 b_2 + a_2 b_1) i \right) y_j \end{aligned}$$

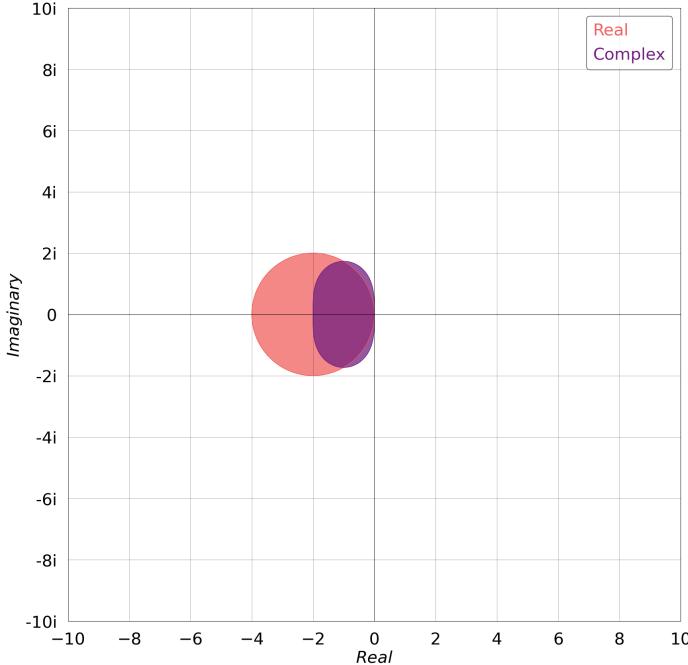
$$y_{j+1} \in \mathbb{R} \implies \text{Im}(y_{j+1}) = 0 \implies b_1 + b_2 + a_1 b_2 + a_2 b_1 = 0 \implies (1 + a_1)b_2 + (1 + a_2)b_1 = 0$$

$$z_1 + z_2 = h \in \mathbb{R} \implies a_1 + a_2 + (b_1 + b_2)i = h \implies b_1 + b_2 = 0 \implies b_1 = -b_2$$

$$(1 + a_1)b_2 + (1 + a_2)b_1 = 0 \implies (1 + a_1)b_2 - (1 + a_2)b_2 = 0 \implies b_2 = 0 = b_1 \text{ or } a_1 = a_2$$

Thus, $z_1, z_2 \in \mathbb{C} \implies a_1 = a_2 \text{ and } b_1 = -b_2$

$$\therefore z_1 = a_1 + b_1 i \text{ and } z_2 = a_1 - b_1 i \implies z_2 = \bar{z}_1 \quad \square$$



Appendix B.8 contains the code for this graph.

As $b \in \mathbb{R}$, $s_c(\lambda, h) \geq s_r(\lambda, h)$

This means the stability region for the complex 2-step method is smaller than that of the real 2-step method; there are less values of λh for which the complex 2-step method is stable.

S_c is smaller than S_r

The diagram above illustrates the case where $b = a = \frac{\lambda h}{2}$

In this case, we have

$$s_r(\lambda, h) = 1 + \lambda h + \frac{(\lambda h)^2}{4} = M(y) + \mathcal{O}((\lambda h)^2)$$

while

$$s_c(\lambda, h) = 1 + \lambda h + \frac{(\lambda h)^2}{2} = M(y) + \mathcal{O}((\lambda h)^3)$$

Euler's Forward 2-Step \mathbb{R}

$$\begin{aligned} y_{j+1} &= \left(1 + \frac{\lambda h}{2}\right)^2 y_j \\ \implies s_r(\lambda, h) &= 1 + \lambda h + \frac{(\lambda h)^2}{4} \end{aligned}$$

Euler's Forward 2-Step \mathbb{C}

$$\begin{aligned} y_{j+1} &= (1 + z)(1 + \bar{z}) y_j \\ &= \left(1 + \frac{\lambda h}{2} + bi\right)\left(1 + \frac{\lambda h}{2} - bi\right) y_j \\ &= \left(\left(1 + \frac{\lambda h}{2}\right)^2 + b^2\right) y_j \\ \implies s_c(\lambda, h) &= 1 + \lambda h + \frac{(\lambda h)^2}{4} + b^2 \\ \implies s_c(\lambda, h) &= s_r(\lambda, h) + b^2 \end{aligned}$$

The complex 2-step method has a higher order of accuracy than the real 2-step method, with the trade-off of a smaller stability region.

We can do the same kind of analysis for other Numerical Methods.

The same derivations and diagrams for the Backward Euler and Runge-Kutta 4 methods follow.

5.3.2 Euler's Backward 2-Step

Theorem 5.2. *Euler's Backward Complex Step Pairs must be Conjugate for $\lambda \in \mathbb{R}$*

For the exponential decay problem, $y = e^{\lambda t}$, let $\lambda \in \mathbb{R}$

Let $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$ be a complex step pair for Euler's Backward 2-Step Method. Then we must have $z_2 = \bar{z}_1$

Proof:

$$\lambda \in \mathbb{R} \implies y_j, y_{j+1} \in \mathbb{R}$$

Euler's Backward 2-Step Method is given by:

$$\begin{aligned} y_{j+1} &= s(\lambda, z_1)s(\lambda, z_2) y_j \\ &= \left(\frac{1}{1 - z_1} \right) \left(\frac{1}{1 - z_2} \right) y_j \\ &= \left(\frac{1}{(1 - z_1)(1 - z_2)} \right) y_j \end{aligned}$$

$$\begin{aligned} (1 - z_1)(1 - z_2) &= (1 - a_1 - b_1 i)(1 - a_2 - b_2 i) \\ &= [1 - a_1 - a_2 + a_1 a_2 - b_1 b_2] + [a_1 b_2 + a_2 b_1 - b_1 - b_2] i \\ &= [(a_1 - 1)(a_2 - 1) - b_1 b_2] + [b_1(a_2 - 1) + b_2(a_1 - 1)] i \end{aligned}$$

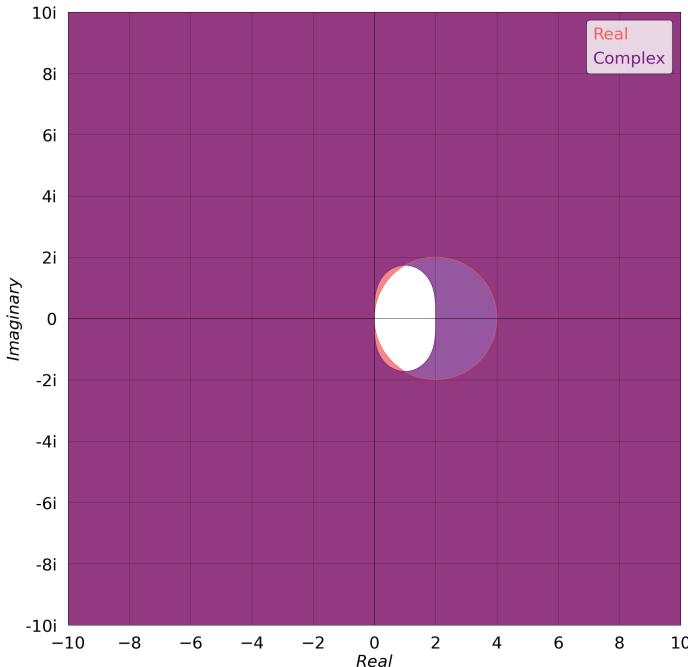
$$y_{j+1} \in \mathbb{R} \implies \text{Im}(y_{j+1}) = 0 \implies b_1(a_2 - 1) + b_2(a_1 - 1) = 0$$

$$z_1 + z_2 = h \in \mathbb{R} \implies a_1 + a_2 + (b_1 + b_2)i = h \implies b_1 + b_2 = 0 \implies b_1 = -b_2$$

$$b_1(a_2 - 1) + b_2(a_1 - 1) = 0 \implies b_1(a_2 - 1) - b_1(a_1 - 1) = 0 \implies b_1 = 0 = b_2 \text{ or } a_1 = a_2$$

Thus, $z_1, z_2 \in \mathbb{C} \implies a_1 = a_2 \text{ and } b_1 = -b_2$

$$\therefore z_1 = a_1 + b_1 i \text{ and } z_2 = a_1 - b_1 i \implies z_2 = \bar{z}_1 \quad \square$$



Euler's Backward 2-Step \mathbb{R}

$$\begin{aligned} y_{j+1} &= \left(\frac{1}{1 - \frac{\lambda h}{2}} \right)^2 y_j \\ \implies s_{\mathbb{R}}(\lambda, h) &= \frac{1}{1 - \lambda h + \frac{(\lambda h)^2}{4}} \end{aligned}$$

Euler's Backward 2-Step \mathbb{C}

$$\begin{aligned} y_{j+1} &= \left(\frac{1}{1 - \frac{\lambda h}{2} + bi} \right) \left(\frac{1}{1 - \frac{\lambda h}{2} - bi} \right) y_j \\ \implies s_{\mathbb{C}}(\lambda, h) &= \frac{1}{1 - \lambda h + \frac{(\lambda h)^2}{4} + b^2} \end{aligned}$$

Again, $b \in \mathbb{R}$, so this time,
 $s_{\mathbb{C}}(\lambda, h) \leq s_{\mathbb{R}}(\lambda, h) \implies S_{\mathbb{C}}$ is larger than $S_{\mathbb{R}}$

Appendix B.8 contains the code for this graph.

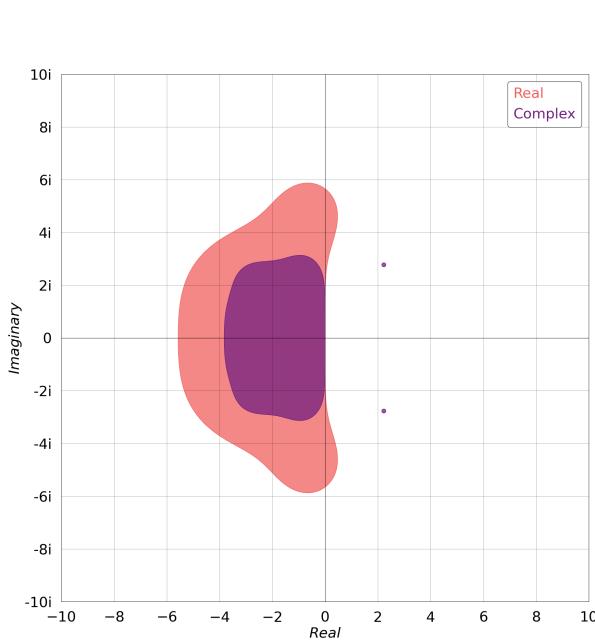
5.3.3 Runge-Kutta 4 2-Step

Conjecture 1. *Runge-Kutta 4 Complex Step Pairs must be Conjugate for $\lambda \in \mathbb{R}$*

This is left as a conjecture for now, as the expansion we did in the other two proofs is much more complicated for the Runge-Kutta 4 method.

The proof is left as an exercise in futility for the reader.

Have a look at Appendix A.2 for the full expansion of the stability function for the Runge-Kutta 4 method, if long algebraic expressions are your flavour of masochism.



Runge-Kutta 4 2-Step \mathbb{R}

$$\begin{aligned} y_{j+1} &= \left(1 + \frac{\lambda h}{2} + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \frac{(\lambda h)^4}{24}\right)^2 y_j \\ y_{j+1} &= \left(1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \frac{(\lambda h)^4}{24} + \frac{(\lambda h)^5}{128} + \frac{5(\lambda h)^6}{4608} + \frac{(\lambda h)^7}{9216} + \frac{(\lambda h)^8}{147456}\right) y_j \\ \implies s_{\mathbb{R}}(\lambda, h) &= 1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \frac{(\lambda h)^4}{24} + \frac{(\lambda h)^5}{128} + \frac{5(\lambda h)^6}{4608} + \frac{(\lambda h)^7}{9216} + \frac{(\lambda h)^8}{147456} \end{aligned}$$

Runge-Kutta 4 2-Step \mathbb{C}

$$y_{j+1} = \left(1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24}\right) \left(1 + \bar{z} + \frac{\bar{z}^2}{2} + \frac{\bar{z}^3}{6} + \frac{\bar{z}^4}{24}\right) y_j$$

Appendix B.8 contains the code for this graph.

$$\implies s_{\mathbb{C}}(\lambda, h) = 1 + \lambda h + \frac{\lambda h^2}{2} + \frac{\lambda h^3}{6} + \frac{\lambda h^4}{24} + \frac{\lambda h^5}{128} + \frac{5\lambda h^6}{4608} + \frac{\lambda h^7}{9216} + \frac{\lambda h^8}{147456} + \frac{b^2 \lambda h^3}{48} + \frac{b^2 \lambda h^4}{128} + \frac{b^2 \lambda h^5}{768} + \frac{b^2 \lambda h^6}{9216} + \frac{b^4 \lambda h}{24} - \frac{b^4 \lambda h^2}{96} + \frac{b^4 \lambda h^3}{192} + \frac{b^4 \lambda h^4}{1536} - \frac{b^6}{72} + \frac{b^6 \lambda h}{144} + \frac{b^6 \lambda h^2}{576} + \frac{b^8}{576}$$

5.4 Varying b for Complex Conjugate Step Pairs

As mentioned earlier, we need not only focus on the case where $b = \frac{\lambda h}{2}$

We can vary b and see how the stability regions change.

In the cases below, we have preserved $a = \frac{\lambda h}{2}$

Consequently, these are still complex conjugate step pairs.

Videos showing the stability regions for varying b values can be found in the *GitHub Repository*[3] for this project.

These videos were created frame-by-frame using the script in Appendix B.9.

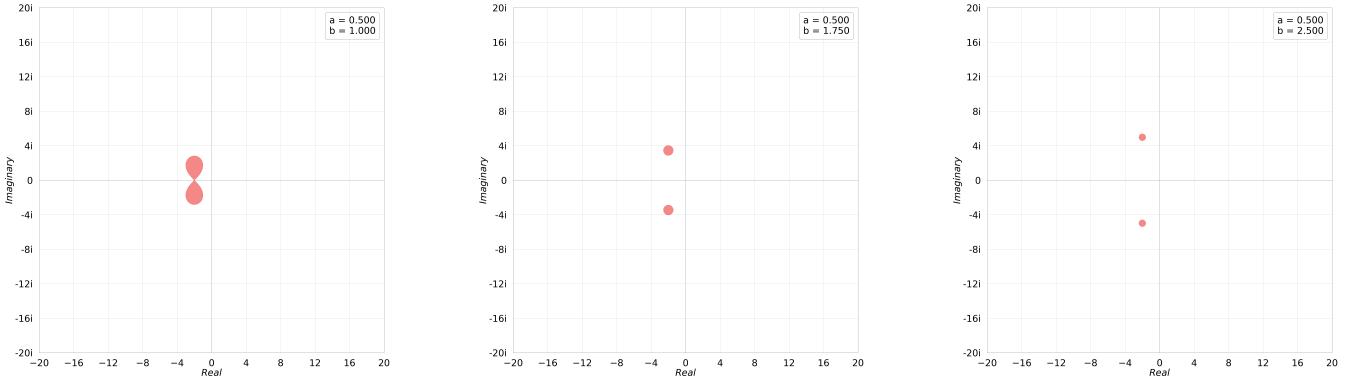
The frames were stitched together using the script in Appendix B.10.

Below are some of the video frames for each method.

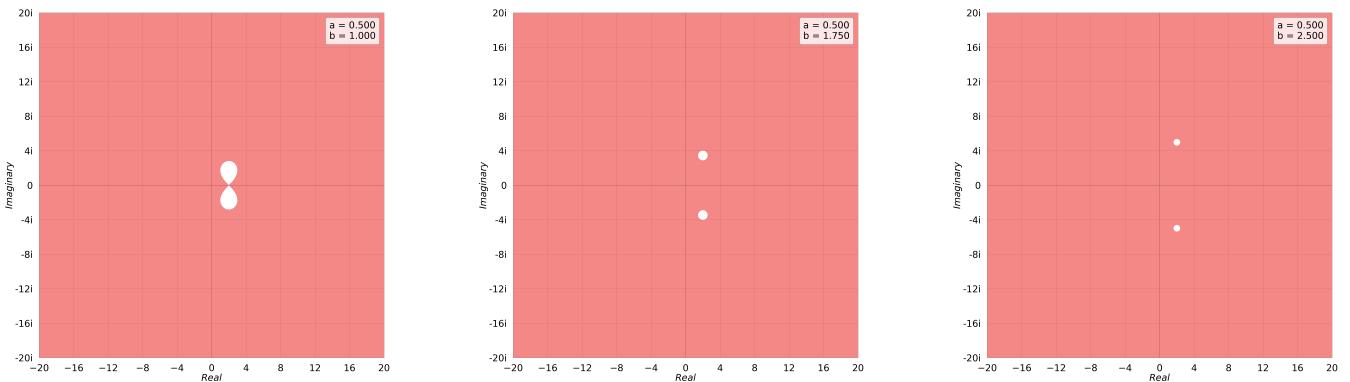
Observations:

- The stability region for complex conjugate pairs is always symmetric across the real axis.
- As b increases, the stability region shrinks for explicit methods and grows for implicit methods. This can be seen quite simply in the $s_c(\lambda, h)$ equations for Euler's Forward and Backward methods.
- The stability region for Euler's Backward Method is the complement of that for Euler's Forward Method, after it has been reflected across the imaginary axis.

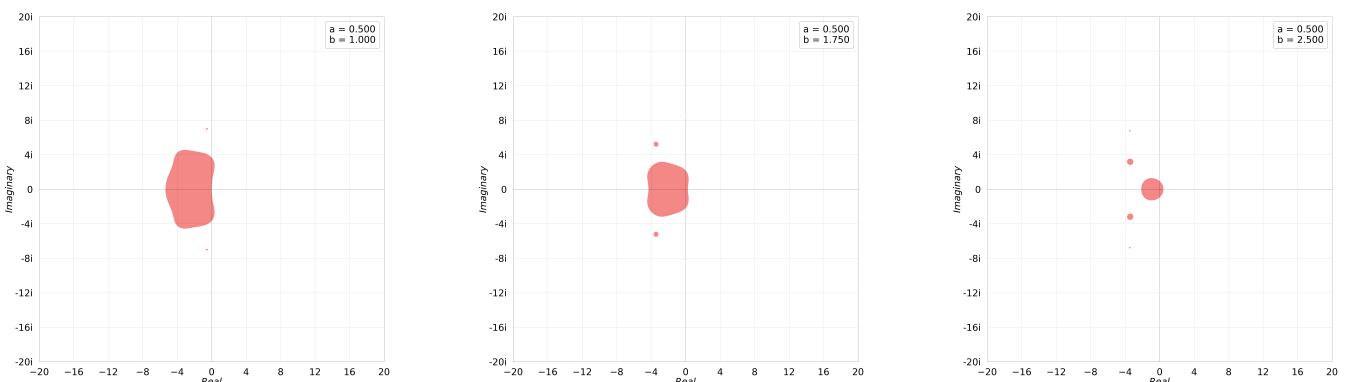
5.4.1 Euler's Forward



5.4.2 Euler's Backward



5.4.3 Runga-Kutta 4



5.5 Varying a

We can also vary a and see how the stability regions change.

This falls outside the scope of complex conjugate step pairs.

Consequently, these stability regions can only be interpreted for $\lambda \in \mathbb{C} \setminus \mathbb{R}$ (If you believe Conjecture 1).

In the cases below, we have set $b = \frac{\lambda h}{2}$

Videos showing the stability regions for varying a values can be found in the *GitHub Repository*[3] for this project.

The same process was followed as in the previous section:

The videos were created frame-by-frame using the script in Appendix B.9.

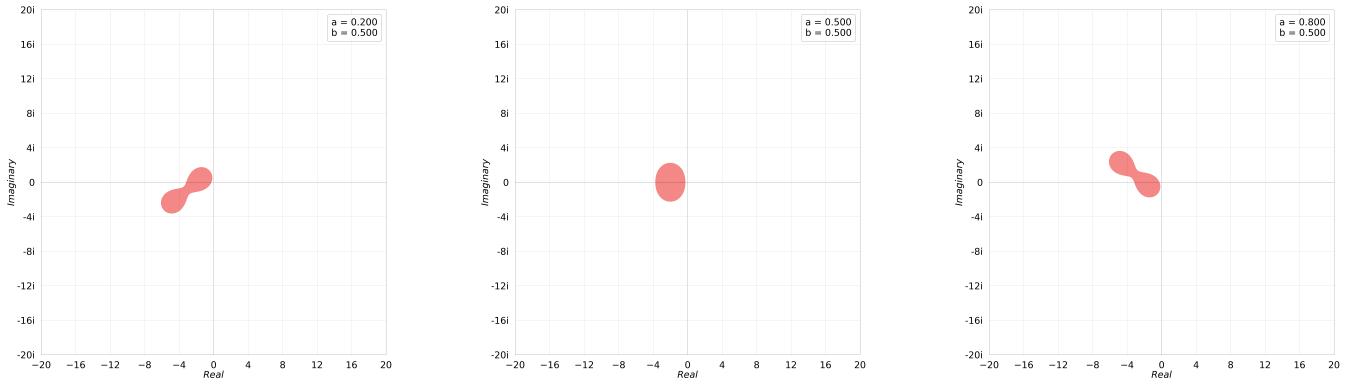
The frames were stitched together using the script in Appendix B.10.

Below are some of the video frames for each method.

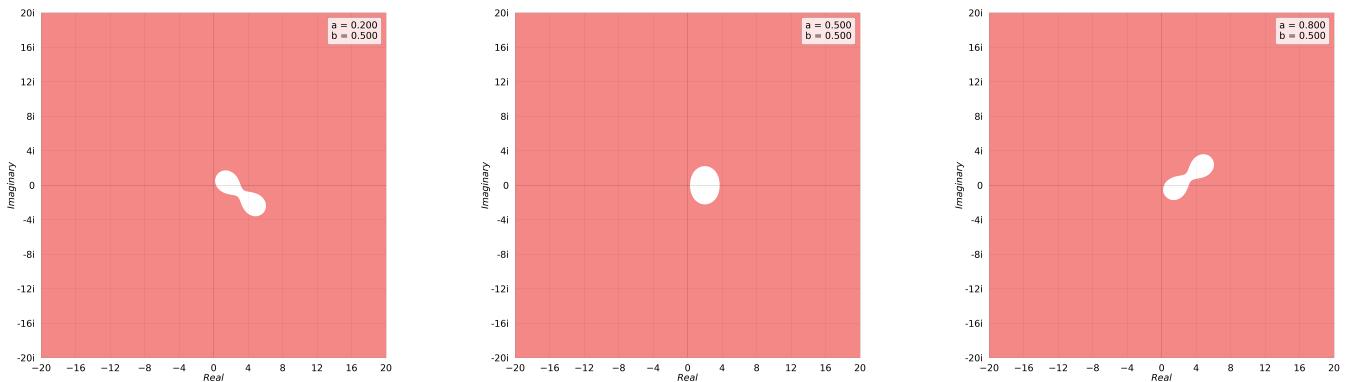
Observations:

- The stability region for varying a values is not symmetric across the real axis, except for the case where $a = 0.5$.
- The stability regions for $a = 0.5 \pm \alpha$ are symmetric to each other across the real axis for any $|\alpha| < 0.5$.
- The stability region for Euler's Backward Method is the complement of that for Euler's Forward Method, after it has been reflected across the imaginary axis.

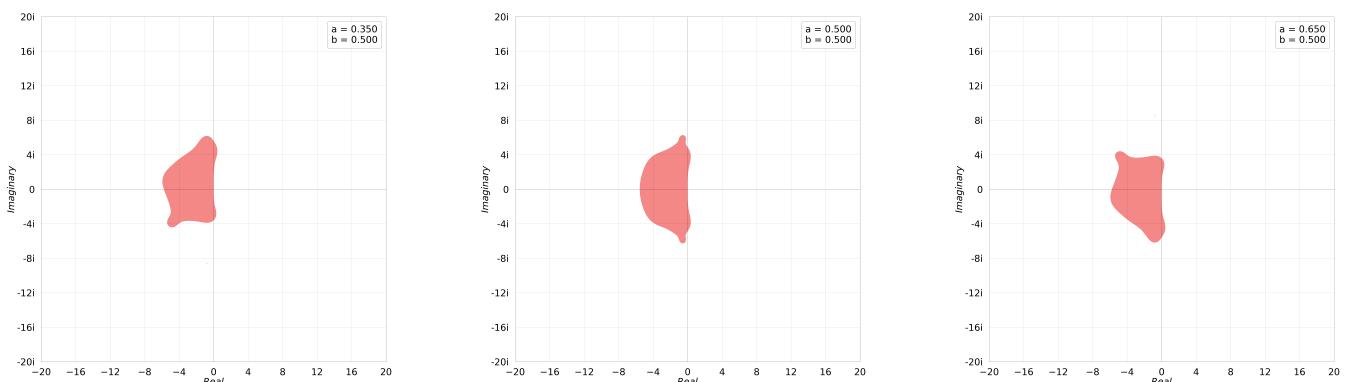
5.5.1 Euler's Forward



5.5.2 Euler's Backward



5.5.3 Runge-Kutta 4



6 Conclusions

6.1 Summary

In this paper, we explored the stability of numerical methods.

We saw that the stability is a crucial property that determines the utility of a numerical method.

We outlined the method for isolating the stability function of a numerical method.

We established the concept of a stability region as the set of pairs (λ, h) for which the numerical method is stable, and interpreted this set as a subset of the complex plane.

We interpreted this region for different numerical methods, showing how methods converge to the exact solution for pairs landing within, and diverge for pairs landing outside.

We also had a look at the use of complex time steps for various numerical methods.

We analysed the behaviour of stability regions for different complex time steps.

We saw evidence that the stability region shrinks as the error decreases.

We also saw that the stability region is not always symmetric about the real axis.

6.2 Personal Reflection

With this project I undertook a few personal challenges.

I succeeded in most with the following outcomes:

- I have learned the tricks of graphing in python, and become more appreciative of the language overall.
- I have figured out methods for making graphical videos, and have learned how to use them to illustrate mathematical concepts.
- I have refined my use of LaTeX, and learned how to use it for more complex documents.
- I have expanded my knowledge of numerical methods, and have a better understanding of the importance of stability.
- I have stitched together knowledge from various modules, and applied it to a single project.
- I have developed a greater appreciation for the mathematics researcher, and the work that goes into a single paper.
- I have learned the basics of working with GitHub. Hello, world. Watch out, world.

6.3 Future Work

There are a few areas that could be expanded upon in future work.

The python written for this project could be compiled into a package for use by others.

This would allow for the easy generation of stability region graphs for various numerical methods.

With the addition of a few more functions, this could be made as easy as entering the equation of the method.

This could be extended to include more numerical methods and toy problems.

It may be particularly useful and intuition building to plot regions of valid h for a given λ , or vice versa.

It would be interesting to explore the stability of more complicated numerical methods.

For example, the stability of multi-step methods.

A small exploration of this was attempted during the project, but didn't go very far.

Have a look at Appendix A.1 for more information.

The paper by *George, Yung and Mangan*[2] looked beyond 2-step complex paths to multi-step paths.

For example, a 3-step path comprised of substeps complex-real-complex adding to an overall real step of size h .

This could be explored further, and the stability regions for these paths could be analysed.

This of course means that the complexity of the stability equation would grow quickly with the number of intermittent steps.

6.4 Acknowledgements

Thank you to Dr. Kirk Soodhalter for taking an extra student under his wing, halfway through an academic year, and reworking a nightmare project into an enjoyable one.

Without your patience and guidance, this project would not have been possible.

Thank you for nurturing my curiosity and for continually challenging me to think deeper.

Thank you to Dr. Nicolas Mascot for helping a lost sheep to find a new shepherd, for organising the transfer of supervision, the rescheduling of deadlines, and for being so understanding.

Thank you to all of my lecturers and tutors in the Department of Mathematics at Trinity College Dublin, for providing me with the tools and knowledge to undertake this project, and ultimately, my degree.

Thank you to my partner, who used my ramblings as her nightly lullaby, told me it was all too complex, and questioned my stability.

You've followed me through every spiral, and didn't diverge.

Thank you for your support, your patience, and for always being there.

Thank you to my brother, who was all too often called for input on a whiteboard he didn't understand, and graphs he couldn't interpret.

Lastly, thank you to my parents for parenting.

You didn't do half bad, guys.

I love you both, and don't show it enough.

I promise no maths chat at the dinner table for a while.

References

- [1] Lloyd N. Trefethen, *The definition of numerical analysis*, Cornell University, 1992.
- [2] Jithin D. George, Samuel Y. Jung, and Niall M. Mangan, *Walking into the complex plane to ‘order’ better time integrators*, <https://arxiv.org/abs/2110.04402>, 2021.
- [3] The GitHub repository for this project, <https://github.com/CianJDuggan/Capstone-Project>
- [4] The Abysmal Kramer-Butler Method, found in the notes of Dr John butler of TU Dublin, https://john-s-butler-dit.github.io/NumericalAnalysisBook/Chapter%2005%20-%20IVP%20Consistent%20Convergence%20Stability/503_Stability.html#step-abysmal-kramer-butler-method

Appendices

A Additional Explorations

A.1 Numerical Method: 2-step Abysmal Kramer-Butler Method

A look was taken beyond 1-step numerical methods.

The 2-step Abysmal Kramer-Butler [4] method was designed to be unstable for the sake of demonstration.

The algorithm is as follows:

$$y_{j+1} = y_{j-1} + h(4\phi(t_j, y_j) - 2\phi(t_{j-1}, y_{j-1}))$$

As we know the exact solution to our Exponential Decay problem is $y(t) = e^{\lambda t}$, we know $\phi(t, y) = \lambda y$.

This gives:

$$y_{j+1} = y_{j-1} + h(4\lambda y_j - 2\lambda y_{j-1})$$

Simplified:

$$y_{j+1} = 4\lambda h y_j + (1 - 2\lambda h) y_{j-1}$$

We need this in the form $y_{j+1} = s(\lambda h)y_j$ to find the stability region.

The author could not manage this.

If this is your poison, good luck.

A.2 Runge-Kutta 4 Complex Step Pairs Conjecture

This is part of the conjecture that the Runge-Kutta 4 method has conjugate step pairs when $\lambda \in \mathbb{R}$. Here follows the expansion of $s(z_1, h)s(z_2, h)$ for Runge-Kutta 4 with complex step pairs $z_1 = a + bi$ and $z_2 = c + di$:

$$\begin{aligned}
& \text{Real: } 1 + \frac{a^4 c^4}{576} + \frac{a^4 c^3}{96} - \frac{a^4 c^2 d^2}{48} + \frac{a^4 c^2}{48} - \frac{a^4 c d^2}{48} + \frac{a^4 c}{24} + \frac{a^4 d^4}{576} - \frac{a^4 d^2}{48} + \frac{a^4}{24} - \frac{a^3 b c^3 d}{36} - \frac{a^3 b c^2 d}{12} + \frac{a^3 b c d^3}{36} - \frac{a^3 b c d}{6} + \frac{a^3 b d^3}{36} - \frac{a^3 b d}{6} + \\
& \frac{a^3 c^4}{144} + \frac{a^3 c^3}{36} - \frac{a^3 c^2 d^2}{24} + \frac{a^3 c^2}{12} - \frac{a^3 c d^2}{6} + \frac{a^3 c}{12} + \frac{a^3 d^4}{144} - \frac{a^3 d^2}{12} + \frac{a^3}{6} - \frac{a^2 b^2 c^4}{96} - \frac{a^2 b^2 c^3}{24} + \frac{a^2 b^2 c^2 d^2}{16} - \frac{a^2 b^2 c^2}{8} + \frac{a^2 b^2 c d^2}{8} - \frac{a^2 b^2 c}{4} - \\
& \frac{a^2 b^2 d^4}{96} + \frac{a^2 b^2 d^2}{8} - \frac{a^2 b^2}{4} - \frac{a^2 b c^3 d}{12} - \frac{a^2 b c^2 d}{12} + \frac{a^2 b c d^3}{12} - \frac{a^2 b c d}{12} + \frac{a^2 b d^3}{12} - \frac{a^2 b d}{2} + \frac{a^2 c^4}{48} + \frac{a^2 c^3}{12} - \frac{a^2 c^2 d^2}{8} + \frac{a^2 c^2}{4} - \frac{a^2 c d^2}{2} + \frac{a^2 c}{2} + \frac{a^2 d^4}{48} - \\
& \frac{a^2 d^2}{4} + \frac{a^2}{2} + \frac{a b^3 c^3 d}{36} + \frac{a b^3 c^2 d}{12} - \frac{a b^3 c^3 d}{36} + \frac{a b^3 c d}{6} - \frac{a b^3 d^3}{36} + \frac{a b^3 d}{6} - \frac{a b^2 c^4}{48} - \frac{a b^2 c^3}{12} + \frac{a b^2 c^2 d^2}{8} - \frac{a b^2 c^2}{4} + \frac{a b^2 c d^2}{4} - \frac{a b^2 c}{2} - \frac{a b^2 d^4}{48} + \\
& \frac{a b^2 d^2}{4} - \frac{a b^2}{2} - \frac{a b c^3 d}{6} - \frac{a b c^2 d}{2} + \frac{a b c d^3}{6} - a b c d + \frac{a b d^3}{6} - a b d + \frac{a c^4}{24} + \frac{a c^3}{6} - \frac{a c^2 d^2}{4} + \frac{a c^2}{2} - \frac{a c d^2}{2} + a c + \frac{a d^4}{24} - \frac{a d^2}{2} + a + \frac{b^4 c^4}{576} + \\
& \frac{b^4 c^3}{144} - \frac{b^4 c^2 d^2}{96} + \frac{b^4 c^2}{48} - \frac{b^4 c d^2}{48} + \frac{b^4 c}{24} + \frac{b^4 d^4}{576} - \frac{b^4 d^2}{48} + \frac{b^4}{24} + \frac{b^3 c^3 d}{36} + \frac{b^3 c^2 d}{12} - \frac{b^3 c d^3}{36} + \frac{b^3 c d}{6} - \frac{b^3 d^3}{36} + \frac{b^3 d}{6} - \frac{b^2 c^4}{48} - \frac{b^2 c^3}{12} + \frac{b^2 c^2 d^2}{8} - \\
& \frac{b^2 c^2}{4} + \frac{b^2 c d^2}{4} - \frac{b^2 c}{2} - \frac{b^2 d^4}{48} + \frac{b^2 d^2}{4} - \frac{b^2}{2} - \frac{b c^3 d}{6} - \frac{b c^2 d}{2} + \frac{b c d^3}{6} - b c d + \frac{b d^3}{6} - b d + \frac{c^4}{24} + \frac{c^3}{6} - \frac{c^2 d^2}{4} + \frac{c^2}{2} - \frac{c d^2}{2} + c + \frac{d^4}{24} - \frac{d^2}{2}
\end{aligned}$$

Imaginary:

B Code

B.1 Exact Solution for Exponential Decay

Found at '[/Python/Exponential Decay/Exact.py](#)' in the repository [3].

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
```

```
cmap = mpl.colormaps[ 'magma_r' ]  
colors = cmap(np.linspace(0, 1, 10))
```

```
def plot_exponential_function(domain, grid, h, lam):
```

```

x = grid/2
t = np.arange(0, x, 1/1000)

y = np.exp(lam * t)

if domain == 'real':
    plt.figure(figsize=(grid, grid), facecolor='white', edgecolor='black')
    ax = plt.gca()
    ax.plot(t, y, color=colors[3])
    ax.set_ylabel('$y$', color='black', fontsize=15)
    ax.set_facecolor('white')
    plt.axhline(0, color='black', linewidth=0.75) # y-axis
    plt.axvline(0, color='black', linewidth=0.75) # x-axis
    # Set border (spines) linewidth
    for spine in ax.spines.values():
        spine.set_linewidth(0.75)
        spine.set_color('black')

    plt.grid(which='major', linestyle='-', linewidth='0.5', color='black')
    ax.set_xlim(-2, 3)
    ax.set_ylim(0, x)
    ax.set_aspect(aspect='equal') # Set the aspect ratio to 1:1
    plt.xticks(np.arange(-2, 4, 1), fontsize=15, color='black')
    plt.yticks(np.arange(0, int(x) + 1, 1), fontsize=15, color='black')
    ax.set_xlabel('$t$', color='black', fontsize=15)
    plt.legend(labels=[r'$\lambda$'.format(lam)], labelcolor=['black', 'black'], face

elif domain == 'complex':
    fig = plt.figure(figsize=(10, 10), facecolor='white', edgecolor='black')
    ax = fig.add_subplot(111, projection='3d', facecolor='white')
    ax.view_init(elev=22.5, azim=315, roll=0)
    ax.plot(t, y.real, y.imag, color=colors[3], linewidth=1.5)
    ax.set_xlabel('$t$', color='black')
    ax.set_ylabel('$\text{Re}(y)$', color='black')
    ax.set_zlabel('$\text{Im}(y)$', color='black')
    plt.legend(labels=[r'$\lambda$'.format(lam)], labelcolor=['black', 'black'], face
    # Set background color for axes planes
    ax.xaxis.set_pane_color('white')
    ax.yaxis.set_pane_color('white')
    ax.zaxis.set_pane_color('white')

    # Customize axis colors
    ax.xaxis.line.set_color('black')
    ax.yaxis.line.set_color('black')
    ax.zaxis.line.set_color('black')

    # Set tick colors
    ax.tick_params(axis='x', colors='black')
    ax.xaxis._axinfo['tick'][ 'inward_factor' ] = 0
    ax.xaxis._axinfo['tick'][ 'outward_factor' ] = 0
    ax.tick_params(axis='y', colors='black')
    ax.yaxis._axinfo['tick'][ 'inward_factor' ] = 0
    ax.yaxis._axinfo['tick'][ 'outward_factor' ] = 0
    ax.tick_params(axis='z', colors='black')
    ax.zaxis._axinfo['tick'][ 'inward_factor' ] = 0
    ax.zaxis._axinfo['tick'][ 'outward_factor' ] = 0

    # Set ranges for axes
    ax.set_xlim([t.min(), t.max()])
    ax.set_ylim([y.real.min(), y.real.max()])
    ax.set_zlim([y.imag.min(), y.imag.max()])

    # Set ticks to create square grid boxes

```

```

tick_space = 4
ax.set_xticks(np.arange(t.min(), t.max() + 1, (np.ceil(t.max()) - np.floor(t.min()))/t
ax.set_yticks(np.arange(np.floor(y.real.min()), np.ceil(y.real.max()), (np.ceil(y.real.
ax.set_zticks(np.arange(np.floor(y.imag.min()), np.ceil(y.imag.max()), (np.ceil(y.imag

# Ensure grid lines are drawn on all planes
ax.grid(True)
ax.grid(color='black')
ax.xaxis._axinfo["grid"].update({"color": "black", "linewidth": 0.5})
ax.yaxis._axinfo["grid"].update({"color": "black", "linewidth": 0.5})
ax.zaxis._axinfo["grid"].update({"color": "black", "linewidth": 0.5})

# Define output path
relative_path = os.path.join('..', 'Graphs', 'Exponential-Decay', f'Exact-with-{lam}.png')

plt.savefig(relative_path, bbox_inches='tight', pad_inches=0)
plt.show()

#plot_exponential_function(domain = 'real', grid = 10, h = 1/5, lam = -1)
plot_exponential_function(domain = 'complex', grid = 20, h = 1/5, lam = -0.3+5j)

```

B.2 Basic Stability Regions

Found at ‘/Python/Stability Regions/Graphs/Basic.py’ in the repository [3].

```
import os
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter as formatter

cmap = mpl.colormaps[ 'magma_r' ]
# Take colors at regular intervals spanning the colormap
colors = cmap(np.linspace(0, 1, 10))

# Stability Functions
EF = [ "Euler's-Forward" , lambda z: z + 1]
EB = [ "Euler's-Backward" , lambda z: 1/(1 - z) ]
RK4 = [ "Runge-Kutta-4" , lambda z: (z**4)/24 + (z**3)/6 + (z**2)/2 + z + 1]
stability_functions = [EF, EB, RK4]

def plot_Region(stability_function, grid, points):
    # Generate domain (complex grid)
    x = grid / 2
    real_domain = np.linspace(-x, x, points)
    imag_domain = np.linspace(-x, x, points)
    real, imag = np.meshgrid(real_domain, imag_domain)
    domain = real + 1j * imag

    # Evaluate Stability Region
    name = stability_function[0]
    function = stability_function[1]
    Range = function(domain)
    distance = np.abs(Range)

    # Define the Figure
    plt.figure(figsize=(grid, grid), facecolor='white', edgecolor='black')
    ax = plt.gca()

    # Plot the Stability Region
    plt.contour(real, imag, distance, levels=[1], colors=[colors[3]], linewidths=1) # Boundary
    plt.contourf(real, imag, distance, levels=[0, 1], colors=[colors[3]], alpha=0.75) # Fill

    # Customize Imaginary axis tick labels to include 'i'
    def im_axis(x, pos):
        return f'{x:.0f}i' if x != 0 else '0'
    ax.yaxis.set_major_formatter(formatter(im_axis))
    ax.set_facecolor('white')

    # Plot bold lines for Real and Imaginary axes
    plt.axhline(0, color='black', linewidth=0.75) # Imaginary axis
    plt.axvline(0, color='black', linewidth=0.75) # Real axis

    # Set major grid lines for both Real and Imaginary axes
    plt.grid(which='major', linestyle='-', linewidth='0.5', color='black')
    ax.set_box_aspect(1)
    ax.set_xlim(-x, x)
    ax.set_ylim(-x, x)
    plt.xticks(np.arange(-int(x), int(x) + 1, 1), fontsize=15, color='black')
    plt.yticks(np.arange(-int(x), int(x) + 1, 1), fontsize=15, color='black')

    # Remove tick marks
    plt.tick_params(axis='both', which='both', length=0, pad=10)

    # Set border (spines) linewidth
    for spine in ax.spines.values():
```

```

spine.set_linewidth(0.75)
spine.set_color('black')

# Labelling
plt.xlabel('$Real$', fontsize=15, color='black')
plt.ylabel('$Imaginary$', fontsize=15, color='black')

relative_path = os.path.join('..', '..', 'Graphs', 'Stability-Regions', 'Graphs', 'Real-1')
plt.savefig(relative_path, bbox_inches='tight', pad_inches=0)
plt.show()

# Iterate through each function in Functions
for s in stability_functions:
    plot_Region(s, 10, 2**13)

```

B.3 Exact vs Methods

Found at ‘/Python/Exponential Decay/Exact Vs Method.py’ in the repository [3].

```

import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

cmap = mpl.colormaps[ 'magma' ]
colors = cmap(np.linspace(0, 1, 10))
background = 'white'
foreground = 'black'

EF = [ "Euler's-Forward" , lambda l , h: (l*h) + 1]
EB = [ "Euler's-Backward" , lambda l , h: 1/(1 - (l*h)) ]
RK4 = [ "Runge-Kutta-4" , lambda l , h: ((l*h)**4)/24 + ((l*h)**3)/6 + ((l*h)**2)/2 + (l*h) + 1]

def plot_exponential_function(domain, grid, h_stable, h_unstable, lam, method, pos, red_width):
    # Size and time steps
    t = np.arange(0, grid, 1/1000)

    ## Exact ##
    y = np.exp(lam * t)

    ## Method ##
    # Stable
    t_steps_stable = np.arange(0, grid+1, 1)
    y_steps_stable = method[1](lam, h_stable) ** t_steps_stable
    y_method_stable = np.interp(t, t_steps_stable, y_steps_stable)

    # Unstable
    t_steps_unstable = np.arange(0, grid+1, 1)
    y_steps_unstable = (method[1](lam, h_unstable) ** t_steps_unstable)
    y_method_unstable = np.interp(t, t_steps_unstable, y_steps_unstable)

    ## Plot ##
    if domain == 'real':
        # Instantiate the graph
        plt.figure(figsize=(10, 10), facecolor=background, edgecolor=foreground)
        ax = plt.gca()

        # Exact
        ax.plot(t, y, color=foreground)

        # Stable
        ax.plot(t, y_method_stable, color='green', linestyle='--')
        ax.plot(t_steps_stable, y_steps_stable, color='green', marker='o', markersize=5, linestyle='')

        # Unstable
        ax.plot(t, y_method_unstable, color='red', linestyle='--')
        ax.plot(t_steps_unstable, y_steps_unstable, color='red', marker='o', markersize=5, linestyle='')

        # Graph settings
        ax.set_ylabel('y$', color=foreground, fontsize=15)
        ax.set_facecolor(background)
        plt.axhline(0, color=foreground, linewidth=0.75) # y-axis
        plt.axvline(0, color=foreground, linewidth=0.75) # x-axis
        for spine in ax.spines.values():
            spine.set_linewidth(0.75)
            spine.set_color(foreground)
        ax.set_xlim(0, grid)
        ax.set_ylim(-grid/4, grid/4)
        ax.set_aspect(aspect='equal') # Set the aspect ratio to 1:1
        plt.xticks(np.arange(0, grid+1, 1), fontsize=15, color=foreground)

```

```

plt.yticks(np.arange(-(grid/4), (grid/4)+1, 1), fontsize=15, color=foreground)
ax.set_xlabel('$t$', color=foreground, fontsize=15)
h_stable = round(h_stable, 3)
h_unstable = round(h_unstable, 3)
plt.legend(labels=[r'$\lambda = \{ \} \cdot \text{format}(lam)', r'$h = \{ \} \cdot \text{format}(h_stable)$', r'$h = \{ \} \cdot \text{format}(h_unstable)$'])

elif domain == 'complex':
    # Instantiate the graph
    fig = plt.figure(figsize=(10, 10), facecolor=background, edgecolor=foreground)
    ax = fig.add_subplot(111, projection='3d', facecolor=background)
    ax.view_init(elev=22.5, azim=315, roll=0)

    # Exact
    ax.plot(t, y.real, y.imag, color=foreground, linewidth=1.5)
    # Valid
    ax.plot(t, y_method_stable.real, y_method_stable.imag, color='green', linewidth=2)
    #ax.plot(t_steps_stable, y_steps_stable.real, y_steps_stable.imag, color='green', marker='x')
    # Instable
    ax.plot(t, y_method_unstable.real, y_method_unstable.imag, color='red', linewidth=red)
    #ax.plot(t_steps_unstable, y_steps_unstable.real, y_steps_unstable.imag, color='red', marker='x')

    # Graph settings
    ax.set_xlabel('$t$', color=foreground)
    ax.set_ylabel('$\text{Re}(y)$', color=foreground)
    ax.set_zlabel('$\text{Im}(y)$', color=foreground)
    plt.legend(labels=[r'$\lambda = \{ \} \cdot \text{format}(lam)$', r'$h = \{ \} \cdot \text{format}(h_{\text{stable}})$', r'$h = \{ \} \cdot \text{format}(h_{\text{unstable}})$'])

    ax.xaxis.set_pane_color(background)
    ax.yaxis.set_pane_color(background)
    ax.zaxis.set_pane_color(background)
    ax.xaxis.line.set_color(foreground)
    ax.yaxis.line.set_color(foreground)
    ax.zaxis.line.set_color(foreground)
    ax.tick_params(axis='x', colors=foreground)
    ax.xaxis._axinfo['tick'][ 'inward_factor'] = 0
    ax.xaxis._axinfo['tick'][ 'outward_factor'] = 0
    ax.tick_params(axis='y', colors=foreground)
    ax.yaxis._axinfo['tick'][ 'inward_factor'] = 0
    ax.yaxis._axinfo['tick'][ 'outward_factor'] = 0
    ax.tick_params(axis='z', colors=foreground)
    ax.zaxis._axinfo['tick'][ 'inward_factor'] = 0
    ax.zaxis._axinfo['tick'][ 'outward_factor'] = 0

    ax.set_xlim(t.min(), t.max())

    y_real_max = max(y.real.max(), y_method_stable.real.max(), y_method_unstable.real.max())
    y_real_min = min(y.real.min(), y_method_stable.real.min(), y_method_unstable.real.min())
    ax.set_ylim(y_real_min, y_real_max)

    y_imag_max = max(y.imag.max(), y_method_stable.imag.max(), y_method_unstable.imag.max())
    y_imag_min = min(y.imag.min(), y_method_stable.imag.min(), y_method_unstable.imag.min())
    ax.set_zlim(y_imag_min, y_imag_max)

    ax.set_xticks(np.arange(np.floor(t.min()), np.ceil(t.max()) + 1, grid/5))
    ax.set_yticks(np.arange(np.floor(y_real_min), np.ceil(y_real_max) + 1, np.ceil(y_real_max) - np.floor(y_real_min)))
    ax.set_zticks(np.arange(np.floor(y_imag_min), np.ceil(y_imag_max) + 1, np.ceil(y_imag_max) - np.floor(y_imag_min)))

    ax.grid(True)
    ax.grid(color=foreground)
    ax.xaxis._axinfo["grid"].update({ "color": foreground, "linewidth": 0.5})
    ax.yaxis._axinfo["grid"].update({ "color": foreground, "linewidth": 0.5})
    ax.zaxis._axinfo["grid"].update({ "color": foreground, "linewidth": 0.5})

```

```

relative_path = os.path.join('..', '..', 'Graphs', 'Exponential-Decay', 'Exact-vs-Method')
plt.savefig(relative_path, bbox_inches='tight', pad_inches=0)

plt.show()

# Euler's Forward
# Real
#plot_exponential_function(domain = 'real', grid = 10, h_stable = 1/6, h_unstable = 5/12, lam
# Complex
#plot_exponential_function(domain = 'complex', grid = 300, h_stable = 0.03, h_unstable = 0.05,

# Euler's Backward
# Real
#plot_exponential_function(domain = 'real', grid = 10, h_stable = 1/6, h_unstable = 5/12, lam
# Complex
#plot_exponential_function(domain = 'complex', grid = 300, h_stable = 0.03, h_unstable = 0.05,

# Runge-Kutta 4
# Real
#plot_exponential_function(domain = 'real', grid = 10, h_stable = 0.5, h_unstable = 0.57, lam
# Complex
plot_exponential_function(domain = 'complex', grid = 300, h_stable = 0.03, h_unstable = 0.5875

```

B.4 Stability Magnitude 3D

Found at ‘/Python/Stability Magnitude/Stability Magnitude 3D.py’ in the repository [3].

```
import os
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter as formatter
from matplotlib.colors import LinearSegmentedColormap, Normalize
import matplotlib as mpl

cmap = mpl.colormaps[ 'magma_r' ]
# Take colors at regular intervals spanning the colormap
colors = cmap(np.linspace(0, 1, 10))

# Define stability functions
EF = [ "Euler's-Forward" , lambda z: z + 1]
EB = [ "Euler's-Backward" , lambda z: 1/(1 - z) ]
RK4 = [ "Runge-Kutta-4" , lambda z: (z**4)/24 + (z**3)/6 + (z**2)/2 + z + 1]

def plot_stability_region(method, grid, points, cmap):
    # Generate domain (complex grid)
    x = grid/2
    real_domain = np.linspace(-x, x, points)
    imag_domain = np.linspace(-x, x, points)
    real, imag = np.meshgrid(real_domain, imag_domain)
    domain = real + 1j * imag

    # Evaluate Stability Region
    name = method[0]
    function = method[1]
    magnitudes = np.abs(function(domain))
    stable_mags = np.where(magnitudes < 1, magnitudes, np.nan)
    unstable_mags = np.where(magnitudes >= 1, magnitudes, np.nan)

    # Define the Figure
    fig = plt.figure(figsize=(grid, grid), facecolor='white', edgecolor='black')
    ax = fig.add_subplot(111, projection='3d', facecolor='white')
    ax.view_init(elev=22.5, azim=22.5, roll=0)

    X, Y, Z = real, imag, stable_mags
    ax.plot_surface(X,Y,Z, edgecolor=colors[3], color=colors[3], linewidth=0.5, alpha=0.3, rco
    ax.contourf(X, Y, Z, zdir='z', offset=0, levels=100, cmap=cmap)
    ax.set_zlim(0, 1)
    ax.set_zticks(np.arange(0, 1.2, 0.2))

    # Set background color for axes planes
    ax.xaxis.set_pane_color('white')
    ax.yaxis.set_pane_color('white')
    ax.zaxis.set_pane_color('white')

    # Customize axis colors
    ax.xaxis.line.set_color('black')
    ax.yaxis.line.set_color('black')
    ax.zaxis.line.set_color('black')

    # Set tick colors
    ax.tick_params(axis='x', colors='black')
    ax.xaxis._axinfo[ 'tick' ][ 'inward_factor' ] = 0
    ax.xaxis._axinfo[ 'tick' ][ 'outward_factor' ] = 0
    ax.tick_params(axis='y', colors='black')
    ax.yaxis._axinfo[ 'tick' ][ 'inward_factor' ] = 0
    ax.yaxis._axinfo[ 'tick' ][ 'outward_factor' ] = 0
    ax.tick_params(axis='z', colors='black')
    ax.zaxis._axinfo[ 'tick' ][ 'inward_factor' ] = 0
```

```

ax.xaxis._axinfo[ 'tick '][ 'outward_factor ']= 0

# Axes
ax.set_xlabel( 'Real', color='black')
ax.set_ylabel( 'Imaginary', color='black')
ax.set_zlabel( 'Magnitude', color='black')
ax.set_xlim(-(x+1), x+1)
ax.set_xticks(np.arange(-x, x+1, 2))
ax.set_ylimits(-(x+1), x+1)
ax.set_yticks(np.arange(-x, x+1, 2))

relative_path = os.path.join( '..', '..', 'Graphs', 'Stability-Magnitude', '3D', f'{name}')
plt.savefig(relative_path, bbox_inches='tight', pad_inches=0)
plt.show()

# Iterate through each function in Functions
plot_stability_region(RK4, 10, 2**12, cmap)

```

B.5 Stability Magnitude Contour

Found at ‘/Python/Stability/Magnitude/Stability Magnitude Color.py’ in the repository [3].

```

import os
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter as formatter
from matplotlib.colors import LinearSegmentedColormap, Normalize
import matplotlib as mpl

cmap = mpl.colormaps[ 'magma_r' ]
# Take colors at regular intervals spanning the colormap
colors = cmap(np.linspace(0, 1, 10))

# Define stability functions
EF = [ "Euler's-Forward" , lambda z: z + 1]
EB = [ "Euler's-Backward" , lambda z: 1/(1 - z) ]
RK4 = [ "Runge-Kutta-4" , lambda z: (z**4)/24 + (z**3)/6 + (z**2)/2 + z + 1]

def plot_stability_region(method, grid, points, cmap):
    # Generate domain (complex grid)
    x = grid/2
    real_domain = np.linspace(-x, x, points)
    imag_domain = np.linspace(-x, x, points)
    real, imag = np.meshgrid(real_domain, imag_domain)
    domain = real + 1j * imag

    # Evaluate Stability Region
    name = method[0]
    function = method[1]
    magnitudes = np.abs(function(domain))
    stable_mags = np.where(magnitudes < 1, magnitudes, np.nan)
    unstable_mags = np.where(magnitudes > 1, magnitudes, np.nan)

    # Define the Figure
    plt.figure(figsize=(grid, grid), facecolor='white', edgecolor='black')
    ax = plt.gca()

    # Plot the Stability Region
    plt.contourf(real, imag, stable_mags, levels=[0, 0.2, 0.4, 0.6, 0.8, 1], cmap=cmap)
    cbar = plt.colorbar(label='Magnitude', format='%.2f', ticks=[0, 0.2, 0.4, 0.6, 0.8, 1])
    # Colorbar
    cbar.ax.tick_params(colors='black', labelsize=15)
    cbar.ax.yaxis.label.set_color('black')
    cbar.ax.yaxis.label.set_size(15)
    # Unstable region
    plt.contourf(real, imag, unstable_mags, levels=1, colors='white')
    # Boundary line
    plt.contour(real, imag, magnitudes, levels=[1], colors='black', linewidths=2)

    # Customize Imaginary axis tick labels to include 'i'
    def im_axis(x, pos):
        return f'{x:.0f}i' if x != 0 else '0'
    ax.yaxis.set_major_formatter(formatter(im_axis))
    ax.set_facecolor('white')

    # Plot bold lines for Real and Imaginary axes
    plt.axhline(0, color='black', linewidth=0.75) # Imaginary axis
    plt.axvline(0, color='black', linewidth=0.75) # Real axis

    # Set major grid lines for both Real and Imaginary axes
    plt.grid(which='major', linestyle='-', linewidth='0.5', color='black')
    ax.set_box_aspect(1)
    ax.set_xlim(-x, x)

```

```

ax.set_xlim(-x, x)
plt.xticks(np.arange(-int(x), int(x) + 1, 1), fontsize=15, color='black')
plt.yticks(np.arange(-int(x), int(x) + 1, 1), fontsize=15, color='black')

# Remove tick marks
plt.tick_params(axis='both', which='both', length=0, pad=10)

# Set border (spines) linewidth
for spine in ax.spines.values():
    spine.set_linewidth(0.75)
    spine.set_color('black')

# Labelling
plt.xlabel('$Real$', fontsize=15, color='black')
plt.ylabel('$Imaginary$', fontsize=15, color='black')

relative_path = os.path.join('..', '..', 'Graphs', 'Stability-Magnitude', 'Color', f'{name}')
plt.savefig(relative_path, bbox_inches='tight', pad_inches=0)
plt.show()

# Iterate through each function in Functions
plot_stability_region(RK4, 10, 2**12, cmap)

```

B.6 Stability Magnitude Solutions

Found at ‘/Python/Stability Magnitude/Exponential Decay.py’ in the repository [3].

```

import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

cmap = mpl.colormaps[ 'magma' ]
colors = cmap(np.linspace(0, 1, 10))
background = 'white'
foreground = 'black'

EF = [ "Euler's-Forward" , lambda l , h: (l*h) + 1]
EB = [ "Euler's-Backward" , lambda l , h: 1/(1 - (l*h)) ]
RK4 = [ "Runge-Kutta-4" , lambda l , h: ((l*h)**4)/24 + ((l*h)**3)/6 + ((l*h)**2)/2 + (l*h) + 1]

def plot_exponential_function(domain, grid, h_values, lam, method):
    # Size and time steps
    t = np.arange(0, grid, 1/1000)

    ## Exact ##
    y = np.exp(lam * t)

    ## Method ##
    t_steps = np.arange(0, grid+1, 1)

    ## Plot ##
    # Instantiate the graph
    plt.figure(figsize=(10, 10), facecolor=background, edgecolor=foreground)
    ax = plt.gca()

    # Exact
    ax.plot(t, y, color=foreground)

    # Method
    for h in h_values:
        index = h_values.index(h)
        y_steps = method[1](lam, h) ** t_steps
        y_method = np.interp(t, t_steps, y_steps)
        ax.plot(t, y_method, color=colors[2*index], linestyle='--')
        ax.plot(t_steps, y_steps, color=colors[2*index], marker='o', markersize=5, linestyle='')

    # Graph settings
    ax.set_ylabel('$y$', color=foreground, fontsize=15)
    ax.set_facecolor(background)
    plt.axhline(0, color=foreground, linewidth=0.75) # y-axis
    plt.axvline(0, color=foreground, linewidth=0.75) # x-axis
    for spine in ax.spines.values():
        spine.set_linewidth(0.75)
        spine.set_color(foreground)
    ax.set_xlim(0, grid)
    ax.set_ylim(-grid/4, grid/4)
    ax.set_aspect(aspect='equal') # Set the aspect ratio to 1:1
    plt.xticks(np.arange(0, grid+1, 1), fontsize=15, color=foreground)
    plt.yticks(np.arange(-(grid/4), (grid/4)+1, 1), fontsize=15, color=foreground)
    ax.set_xlabel('$t$', color=foreground, fontsize=15)
    ax.legend([f'Exact - with $\lambda={lam}$', f'$h={h_values[0]}$',
              f'$h={h_values[1]}$'])

relative_path = os.path.join('..', '..', 'Graphs', 'Stability-Magnitude', 'Exponential-Decay')
plt.savefig(relative_path, bbox_inches='tight', pad_inches=0)

plt.show()

```

```
# Euler's Forward
#plot_exponential_function(domain = 'real', grid = 10, lam = -5, method = EF, h_values = [0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0])
# Euler's Backward
#plot_exponential_function(domain = 'real', grid = 10, lam = -5, method = EB, h_values = [0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0])
# Runge-Kutta 4
plot_exponential_function(domain = 'real', grid = 10, lam = -5, method = RK4, h_values = [0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0])
```

B.7 Runge-Kutta 4 Stability Magnitude

Finding h for $|s_{RK4}| = \lambda h$ when given λ Found at ‘/Python/Stability Magnitude/Rk4 = c.py’ in the repository [3].

```
import numpy as np
from scipy.optimize import fsolve
import math

def objective_function(x, c):
    """
    Objective function to be solved for |sum| = c
    """
    sum_expression = np.sum([(-5*x)**n / math.factorial(n) for n in range(5)])
    return np.abs(sum_expression) - c

def find_solutions(c, initial_guess=0.0):
    """
    Find the roots of the objective function for the given constant c.

    Parameters:
    c (float): The constant value to solve for
    initial_guess (float): Initial guess for the root-finding algorithm

    Returns:
    numpy.ndarray: Array of solutions (x values) for |sum| = c
    """
    solutions = fsolve(objective_function, initial_guess, args=(c,))
    return solutions

# Example usage
c = 0.9
solutions = find_solutions(c)
print(f"Solutions for |sum| = {c}:")
print(solutions)
```

B.8 Real VS Complex Stability Regions

Found at ‘/Python/Stability Regions/Graphs/Real vs Complex Comparison.py’ in the repository [3].

```

import os
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter as formatter
import os

cmap = mpl.colormaps[ 'magma_r' ]
# Take colors at regular intervals spanning the colormap
colors = cmap(np.linspace(0, 1, 10))

# Declare Steps
z_1 = lambda l, a, b: (a + b*1j)*l
z_2 = lambda l, a, b: ((1-a) - b*1j)*l

# Stability Function for Euler's Forward 2-step
EF = ["Euler's-Forward", lambda l, a, b: (1 + z_1(l, a, b)) * (1 + z_2(l, a, b))]

EB = ["Euler's-Backward", lambda l, a, b: 1/(1 - z_1(l, a, b)) * 1/(1 - z_2(l, a, b))]

rk4_poly = lambda x: 1 + x + (x**2)/2 + (x**3)/6 + (x**4)/24
RK4 = ["Runge-Kutta-4", lambda l, a, b: rk4_poly(z_1(l, a, b)) * rk4_poly(z_2(l, a, b))]

# Plot the Stability Region for a given Stability Function
def plot_Region(stability_function, grid, points):

    # Generate domain (complex grid)
    grid_scale = grid / 10
    x = grid / 2
    real_domain = np.linspace(-x, x, points)
    imag_domain = np.linspace(-x, x, points)
    real, imag = np.meshgrid(real_domain, imag_domain)
    domain = real + 1j * imag

    # Evaluate Stability Region
    name = stability_function[0]
    function = stability_function[1]
    # Real
    a_r = 0.5
    b_r = 0
    realRange = function(domain, a_r, b_r)
    realDistance = np.abs(realRange)
    # Complex
    a_c = 0.5
    b_c = 0.5
    complexRange = function(domain, a_c, b_c)
    complexDistance = np.abs(complexRange)

    # Define the Figure
    plt.figure(figsize=(grid, grid), facecolor='white', edgecolor='black')
    ax = plt.gca()

    # Plot the Stability Region
    # Real
    plt.contour(real, imag, realDistance, levels=[1], colors=[colors[3]], linewidths=1) # Boundary
    plt.contourf(real, imag, realDistance, levels=[0, 1], colors=[colors[3]], alpha=0.75) # Inside
    # Complex
    plt.contour(real, imag, complexDistance, levels=[1], colors=[colors[6]], linewidths=1) # Boundary
    plt.contourf(real, imag, complexDistance, levels=[0, 1], colors=[colors[6]], alpha=0.75)

```

```

# Customize Imaginary axis tick labels to include 'i'
def im_axis(x, pos):
    return f'{x:.0f}i' if x != 0 else '0'
ax.yaxis.set_major_formatter(formatter(im_axis))
ax.set_facecolor('white')

# Plot bold lines for Real and Imaginary axes
plt.axhline(0, color='black', linewidth=0.75) # Imaginary axis
plt.axvline(0, color='black', linewidth=0.75) # Real axis

# Set major grid lines for both Real and Imaginary axes
plt.grid(which='major', linestyle='-', linewidth='0.5', color='black')
ax.set_box_aspect(1)
ax.set_xlim(-x, x)
ax.set_ylim(-x, x)
plt.xticks(np.arange(-int(x), int(x) + 1, 1*grid_scale), fontsize=15*grid_scale, color='black')
plt.yticks(np.arange(-int(x), int(x) + 1, 1*grid_scale), fontsize=15*grid_scale, color='black')

# Remove tick marks
plt.tick_params(axis='both', which='both', length=0, pad=10*grid_scale)

# Set border (spines) linewidth
for spine in ax.spines.values():
    spine.set_linewidth(0.75)
    spine.set_color('black')

# Labelling
plt.xlabel('$Real$', fontsize=15*grid_scale, color='black')
plt.ylabel('$Imaginary$', fontsize=15*grid_scale, color='black')

# Legend with "Real" and "Complex" labels in colors matching the plot
plt.legend(labels=['Real', 'Complex'], labelcolor=[colors[3], colors[6]], fontsize=15*grid_scale)

relative_path = os.path.join('..', '..', 'Graphs', 'Stability-Regions', 'Graphs', 'Real-Viscous-Regions')
plt.savefig(relative_path, bbox_inches='tight', pad_inches=0)

plt.show()

plot_Region(EB, 20, 2*13)

```

B.9 Video Frame Generation

Found at ‘/Python/Stability Regions/Videos/Complex Pairs Video Frames.py’ in the repository [3].

```

import os
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter as formatter

cmap = mpl.colormaps[ 'magma_r' ]
# Take colors at regular intervals spanning the colormap
colors = cmap(np.linspace(0, 1, 10))

# Declare Steps
z_1 = lambda l, a, b: a*l + b*1j
z_2 = lambda l, a, b: (1-a)*l - b*1j

# Stability Function for Euler's Forward 2-step
EF = ["Euler's-Forward", lambda l, a, b: (1 + z_1(l, a, b)) * (1 + z_2(l, a, b))]

EB = ["Euler's-Backward", lambda l, a, b: 1/(1 - z_1(l, a, b)) * 1/(1 - z_2(l, a, b))]

rk4_poly = lambda x: 1 + x + (x**2)/2 + (x**3)/6 + (x**4)/24
RK4 = ["Runge-Kutta-4", lambda l, a, b: rk4_poly(z_1(l, a, b)) * rk4_poly(z_2(l, a, b))]

# Define an array of a values
#varied_a = [[0.537], 0.537] #testing
varied_a = [np.arange(0, 1, 0.001), 0.5] # [a-values, b] where 0<a<1 and b is constant
varied_b = [0.5, np.arange(0, 4, 0.005)] # [a, b-values] where a is constant and 0<b<4

# Plot the Stability Region for a given Stability Function
def plot_Region(stability_function, grid, points, aORb, frame_number):

    # Unpack the stability function and varied values for a or b
    if aORb == 'a':
        a_values = varied_a[0]
        b = varied_a[1]
        a = a_values[frame_number]
        subdir = 'Varied-a'
        const = 'b'
        const_val = b
    elif aORb == 'b':
        a = varied_b[0]
        b_values = varied_b[1]
        b = b_values[frame_number]
        subdir = 'Varied-b'
        const = 'a'
        const_val = a
    else:
        raise ValueError('Invalid input for aORb')

    # Generate domain (complex grid)
    grid_scale = grid / 10
    x = grid / 2
    real_domain = np.linspace(-x, x, points)
    imag_domain = np.linspace(-x, x, points)
    real, imag = np.meshgrid(real_domain, imag_domain)
    domain = real + 1j * imag

    # Evaluate Stability Region
    name = stability_function[0]
    function = stability_function[1]
    Range = function(domain, a, b)
    distance = np.abs(Range)

```

```

# Define the Figure
plt.figure(figsize=(grid, grid), facecolor='white', edgecolor='black')
ax = plt.gca()

# Plot the Stability Region
plt.contour(real, imag, distance, levels=[1], colors=[colors[3]], linewidths=1) # Boundary
plt.contourf(real, imag, distance, levels=[0, 1], colors=[colors[3]], alpha=0.75) # Fill

# Customize Imaginary axis tick labels to include 'i'
def im_axis(x, pos):
    return f'{x:.0f}i' if x != 0 else '0'
ax.yaxis.set_major_formatter(formatter(im_axis))
ax.set_facecolor('white')

# Plot bold lines for Real and Imaginary axes
plt.axhline(0, color='black', linewidth=0.75) # Imaginary axis
plt.axvline(0, color='black', linewidth=0.75) # Real axis

# Set major grid lines for both Real and Imaginary axes
plt.grid(which='major', linestyle='-', linewidth='0.5', color='black')
ax.set_box_aspect(1)
ax.set_xlim(-x, x)
ax.set_ylim(-x, x)
plt.xticks(np.arange(-int(x), int(x) + 1, 1*grid_scale), fontsize=15*grid_scale, color='black')
plt.yticks(np.arange(-int(x), int(x) + 1, 1*grid_scale), fontsize=15*grid_scale, color='black')

# Remove tick marks
plt.tick_params(axis='both', which='both', length=0, pad=10*grid_scale)

# Set border (spines) linewidth
for spine in ax.spines.values():
    spine.set_linewidth(0.75)
    spine.set_color('black')

# Labelling
plt.xlabel('$Real$', fontsize=15*grid_scale, color='black')
plt.ylabel('$Imaginary$', fontsize=15*grid_scale, color='black')

# a & b values legend
plt.legend([f'a={a:.3f}\nb={b:.3f}' for a in np.arange(-1, 2, 0.3) for b in np.arange(-1, 2, 0.3)], loc='upper-right', fontsize=15*grid_scale, facecolor='white')

# Save plot as a frame
relative_path = os.path.join('..', '..', '..', 'Graphs', 'Stability-Regions', 'Videos')
video_path = os.path.join(subdir, name, f'{const}={const_val}', 'frames')
output_path = os.path.join(relative_path, video_path)
if not os.path.exists(output_path):
    os.makedirs(output_path)
frame_path = f'{frame_number:04d}.png'
path = os.path.join(output_path, frame_path)
plt.savefig(path, bbox_inches='tight', pad_inches=0)

output_dir = f'/home/puca/University/Senior-Sophister/Capstone/Graphs/Stability-Regions/Videos'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
plt.savefig(f'{output_dir}/{frame_number:04d}.png')
# plt.show()
plt.close()

# Generate frames for Stability Function with multiple a or b values
for i, b in enumerate(varied_b[1]):
    plot_Region(RK4, 40, 1000, 'b', i)

```

B.10 Frame Stitching for Video

Found at ‘/Python/Stability Regions/Videos/Frames-to-Video.py’ in the repository [3].

```
import os
import imageio.v2 as imageio

EF = "Euler's-Forward"
EB = "Euler's-Backward"
RK4 = "Runge-Kutta-4"

method = RK4
varied = "Varied-b"
const = "a=0.5"

# Define the base path using relative paths
base_path = os.path.join('..', '..', '..', 'Graphs', 'Stability-Regions', 'Videos')

# Define the frames directory and the output video path using relative paths
frames_dir = os.path.join(base_path, varied, method, const, 'frames')
output_video_path = os.path.join(base_path, varied, method, const, 'video.mp4')

# Ensure the frames directory exists
if not os.path.exists(frames_dir):
    raise ValueError(f"The-frames-directory-{frames_dir}-does-not-exist.")

# Create the video writer
with imageio.get_writer(output_video_path, fps=30) as writer:
    # Iterate through the frames in the directory
    for frame_number in sorted(os.listdir(frames_dir)):
        # Construct the full path to the frame
        frame_path = os.path.join(frames_dir, frame_number)
        # Read and append the frame to the video
        image = imageio.imread(frame_path)
        writer.append_data(image)

print(f"Video-created-successfully-at-{output_video_path}")
```