coursework_copy

May 1, 2023

0.1 Scientific Computing Coursework [EMAT30008]

0.2 Name: Quincy Sproul

0.3 Student no: 2027185

0.4 Table of contents

1 Introduction

Numerical methods are an essential aspect of scientific computing, providing powerful tools to solve mathematical models and simulations that are too complex to solve analytically. Numerical methods are widely used in various fields of engineering, physics, and mathematics to obtain accurate and efficient solutions to a range of problems.

The software I have developed is a general numerical continuation code that can track limit cycle oscillations of arbitrary ordinary differential equations and steady-states of second-order diffusive partial differential equations. The software is designed to be modular, following the DRY principle, and takes the form of a library, providing one or more functions that take the differential equation (either an ODE or PDE) in a suitable form, the parameter values, and a starting guess for the initial variable values (and period of oscillation if appropriate).

The overall purpose of this software is to provide a reliable and efficient tool for researchers and practitioners in various fields to solve complex mathematical models and simulations. The software is fully tested against a range of inputs and known outputs, and inputs that do not have a solution are handled gracefully. Additionally, the code is appropriately documented, and examples of running the code for both ODEs and PDEs are provided, with no user input required when running the examples.

Throughout the course, we have covered various topics related to numerical methods, such as initial value problems, numerical shooting, code testing, numerical continuation, finite difference methods, the method of lines, implicit methods for linear PDEs, implicit methods for nonlinear

PDEs, and sparse linear algebra. The software I have developed takes into account all of these topics and integrates them into a single, powerful tool for solving complex mathematical models and simulations.

2 Section 2: Brief Summary of the Software (3 pages)

2.0.1 2.1 Summary of Software

The aim of this coursework was to develop a Python package that can solve ordinary and partial differential equations (ODEs and PDEs) with both initial value problems (IVPs) and boundary value problems (BVPs). While existing packages such as `scipy` provide some functionality in this regard, the software has been designed to make the process of finding a solution to differential equations more straightforward and guided by automatically detecting the problem type (ODE vs PDE) based on the function arguments' length and identifying whether it is an IVP or BVP.

The main `ProblemSolver` class forms the backbone of the software, with separate `IVP` and `BVP` classes to handle these two types of problems. The `Problem` class is used to set up the problem by defining the differential equation(s) and initial/boundary conditions, as well as any relevant parameters.

To solve the problem, the `solve()` method of the `ProblemSolver` class is called, which takes several optional arguments to customize the solver's behavior. The method argument determines the ODE solver method used, while `pde_method` is used to select the numerical method for solving PDEs. The `root_finder` argument selects the algorithm used to find roots in the BVP case, while `discretize_method`, `bc_type`, and `matrix_type` are used to customize the discretization method, boundary conditions, and matrix type for BVPs.

The software can handle various types of ODE and PDE problems, including stiff ODEs and systems of equations, as well as nonlinear PDEs. The software has been tested on a range of problems, including several weekly exercises from the labs, and have found that it provides accurate solutions with reasonable computational efficiency.

2.0.2 2.2 Overview of Numerical Software

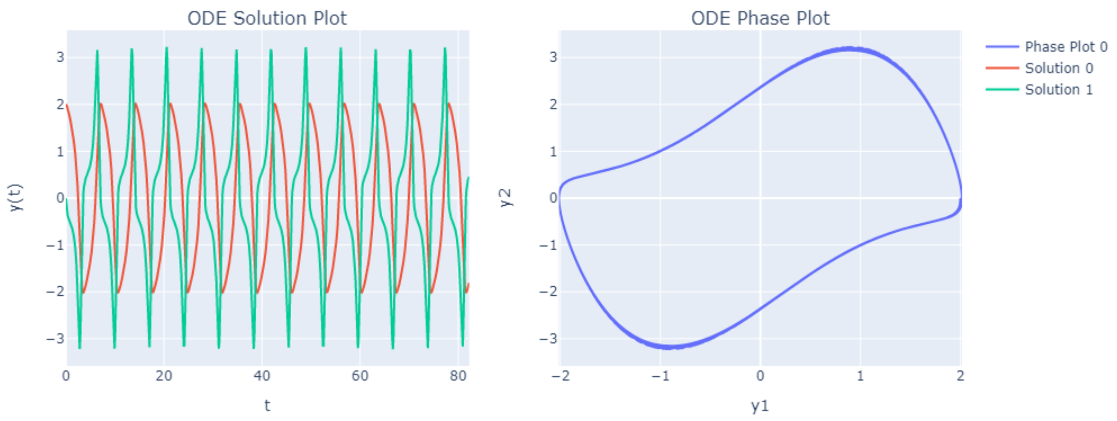
The package is designed to be user-friendly and versatile, with a focus on providing a `ProblemSolver` class that can handle a variety of problems by analyzing user arguments and solving through other modules. The following is an overview of the capabilities of the software and some examples of how it can be used.

2.1 Ordinary Differential Equations The package provides a range of methods for solving ordinary differential equations (ODEs), including Euler's method, the midpoint method, the Runge-Kutta method, and the Dormand-Prince method. Each of the differential solving methods include appropriate step size controlling functionality to prevent the solution from blowing up. The ProblemSolver class can be used to solve ODEs by providing the differential equation, the initial conditions, and the time points.

2.1.1 ODE IVP: Solving the classic Van der Pol oscillator problem

```
[ ]: from src import ProblemSolver as ps
     import numpy as np
```

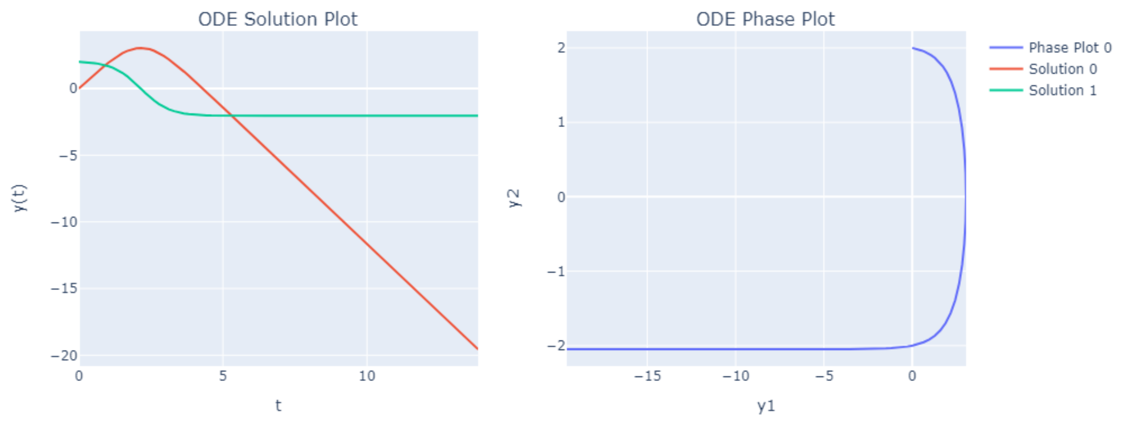
```
# Define the function
def van_der_pol(t, y, mu):
    dydt = [y[1], mu*(1-y[0]**2)*y[1]-y[0]]
    return dydt

# Initial conditions
y0 = np.array([2, 0])

# Time points
t0, tf = 0, 5
Nt = 500

# Parameters
mu = 1.5

# Solve the problem
solver = ps(f = van_der_pol, y0 = y0, t0 = t0, tf = tf, Nt=Nt, args=(mu,))
solution = solver.solve(method="RK45")

# Plot the phase plot
solution.plot(phase_plot=True, width=1000, height=400, margin=dict(l=50, r=50,␣
  ↪b=50, t=50, pad=0))
```



2.1.2 ODE BVP: Solving the Bratu problem using the shooting method

[ ]:
```
from src import ProblemSolver as ps
import numpy as np

# Define the function
def bratu(t, y, lmbda):
    dydx = [y[1], -lmbda*np.exp(y[0])]
    return dydx
```

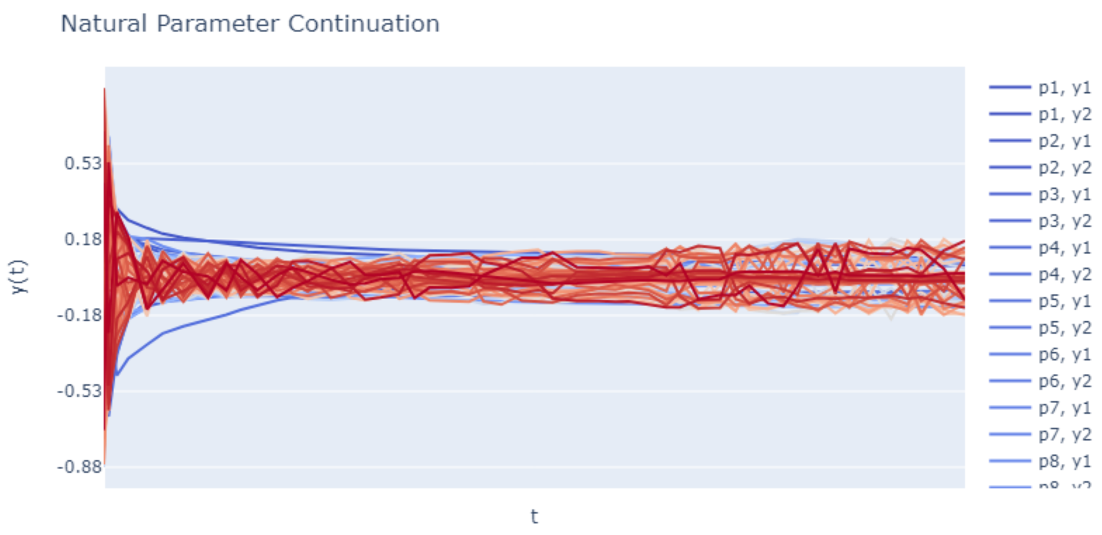
```
# Boundary conditions
def bc(ya, yb):
    return np.array([ya, yb-2])

# Initial guess
y0 = np.array([0, 0])

# Time points
t0 = 0
tf = 1
Nt = 100

# Parameters
lmbda = 0.1

# Solve the problem
solver = ps(f = bratu, y0 = y0, t0 = t0, tf = tf, Nt=Nt, args=(lmbda,), bc=bc)
solution, y0_sol = solver.solve(method="RK45")

# Plot the solution
solution.plot(phase_plot=True, width=1000, height=400, margin=dict(l=50, r=50,␣
 ↪b=50, t=50, pad=0))
```

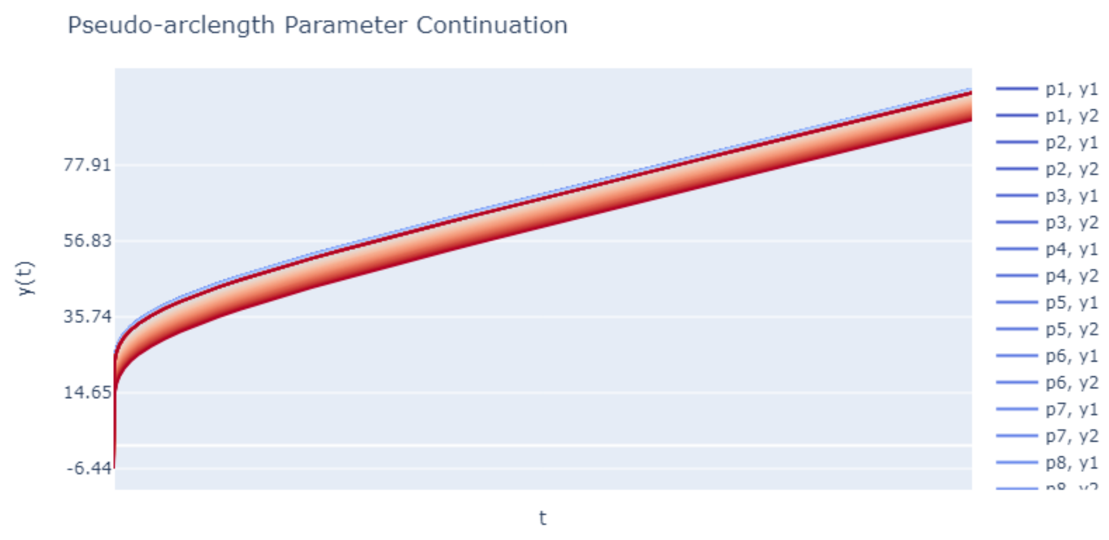


2.1.3 ODE Continuation: Computing the bifurcation diagram of the Duffing oscillator (natural vs pseudo-arclength continuation)

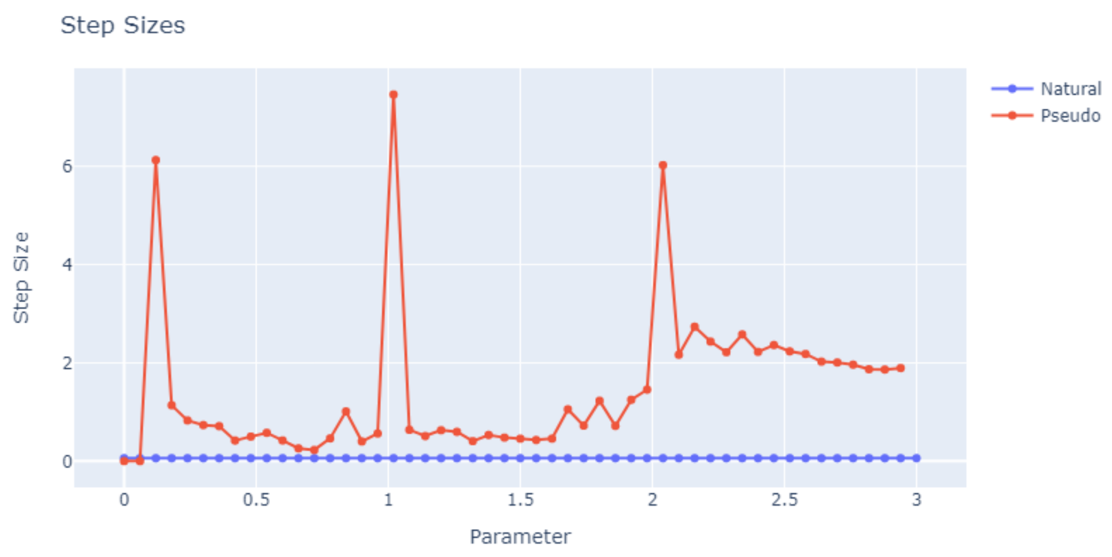
[ ]:
```
from src import ProblemSolver as ps
from src import examples as ex
import numpy as np

example = ex.Duffing()

params, ode, _ = example()
a, b, d, g = params
```

```
# Initial conditions
y0 = np.array([0, 1])

# Time points
t0 = 0
tf = 100
Nt = 50

# Continuation parameters
p_span = [0, 100]
vary_par = 0

# Set the parameters
solver = ps(f = ode, y0 = y0, t0 = t0, tf = tf, Nt=Nt, args=(a, b, d, g))

# Solve the problem using natural continuation
natural_solution, natural_y0s = solver.solve(p_span=p_span,␣
 ↪cont_type="natural", vary_par=vary_par)

# Solve the problem using pseudo continuation
pseudo_solution, pseudo_y0s = solver.solve(p_span=p_span, cont_type="pseudo",␣
 ↪vary_par=vary_par)

natural_solution.plot(title="Natural Parameter Continuation", width=800,␣
 ↪height=400, margin=dict(l=50, r=50, b=50, t=50, pad=0)).
 ↪update_layout(xaxis=dict(tickmode='array', tickvals=[]))
# pseudo_solution.plot(title="Pseudo-arclength Parameter Continuation",␣
 ↪width=800, height=400, margin=dict(l=50, r=50, b=50, t=50, pad=0)).
 ↪update_layout(xaxis=dict(tickmode='array', tickvals=[]))
```





```
[ ]: import plotly.graph_objects as go
     import plotly.express as px
     import numpy as np

     # Extract step sizes for natural continuation method
     s_natural = np.diff(natural_solution.params)
     s_natural = np.concatenate(([s_natural[0]], s_natural))

     # Extract step sizes for pseudo continuation method
     s_pseudo = np.sqrt(np.sum(np.diff(pseudo_solution.y, axis=0)**2, axis=(1, 2)))
     s_pseudo = np.concatenate(([s_pseudo[0]], s_pseudo))

     # Plot the step sizes
     fig = go.Figure()
     fig.add_trace(go.Scatter(x=natural_solution.params, y=s_natural,␣
      ↪mode='lines+markers', name='Natural'))
     fig.add_trace(go.Scatter(x=pseudo_solution.params, y=s_pseudo,␣
      ↪mode='lines+markers', name='Pseudo'))
     fig.update_layout(title="Step Sizes", xaxis_title="Parameter",␣
      ↪yaxis_title="Step Size", width=800, height=400, margin=dict(l=50, r=50,␣
      ↪b=50, t=50, pad=0))
     fig.show()
```



2.3 Partial Differential Equations The package provides methods for solving partial differential equations (PDEs) using finite difference methods, including explicit and implicit methods, Crank-Nicolson method, and method of lines. The ProblemSolver class can be used to solve PDEs by providing the differential equation, the initial conditions, the boundary conditions, and the domain. For example, the following code solves the heat equation u_t = u_xx on the domain [0, 1] x [0, 1] with initial condition u(x, 0) = sin(pi*x) using the Crank-Nicolson method:

2.3.1 PDE IVP: Solving the heat equation using the Crank-Nicolson method

[ ]:
```
from src import ProblemSolver as ps
from src import examples as ex
import numpy as np

example = ex.Heat()

params, _, pde = example()
D = params

# Initial conditions
def ic(x):
    return np.sin(np.pi*x)

# Time points
t0 = 0
tf = 1
dt = 0.01

# Spatial points
dx = 0.01
a, b = 0, 1
```

```
# Set the parameters
solver = ps(q=pde, ic=ic, t0=t0, tf=tf, dt=dt, a=a, b=b, dx=dx, args=(D,), C=0.
 ↪5)

# Solve the problem
solution = solver.solve()

# Plot the solution
solution.plot(title="Heat Equation", width=800, height=400, margin=dict(l=50,␣
 ↪r=50, b=50, t=50, pad=0))
```

2.3.2 PDE BVP: Solving the dynamic Bratu problem

[ ]:
```
from src import ProblemSolver as ps
from src import examples as ex
import numpy as np

example = ex.DynamicBratu()

params, _, pde = example()
D = params

# Define the initial conditions as a function of the spatial dimension (must␣
 ↪return same dimension as solution dimension)
def ic(x):
    return np.zeros_like(x)

# Define the time dependent boundary conditions
def bc(ya, yb, t):
    # Time dependent bc
    return np.array([ya, yb-2*np.exp(-t)])

# Define the problem parameters
x0, xf = 0, 1
t0, tf = 0, 1
Nx = 1000
Nt = 1000
mu = 2

# Solve the problem
solver = ps(q = pde, ic = ic, bc = bc, a = x0, b = xf, t0 = t0, tf = tf, Nx =␣
 ↪Nx, Nt = Nt, args=(mu,),C=0.5)
solution = solver.solve(pde_method="cranknicolson")

# Plot the solution
solution.plot()
```

2.4 Code Profiling

2.4.1 Profiling ODE methods (Euler, IEuler, midpoint, Runge-Kutta, Dormand-Prince)

[ ]:
```python
from src import ProblemSolver as ps
import numpy as np
import cProfile
import pstats

# Define the function
def f(t, y, a, b, c, d):
    dydt = [a*y[0] - b*y[0]*y[1], -c*y[1] + d*y[0]*y[1]]
    return dydt

# Initial conditions
y0 = np.array([3, 1])

# Time points
t0, tf = 0, 100
Nt = 100

# Parameters
a, b, c, d = 1.5, 1, 3, 1

# # Solve the problem
solver = ps(f = f, y0 = y0, t0 = t0, tf = tf, Nt=Nt, args=(a, b, c, d))

# run the profiler
cProfile.run('solution = solver.solve(method="RK45")', 'profiler_output')

# create a Stats object and load the profiler output
stats = pstats.Stats('profiler_output')

# print the function calls and time for solver.solve() method
# stats.strip_dirs().sort_stats('tottime').get_print_list(sel_list=['solve'])
```

3 Section 3: Key Software Design Decisions (4 pages)

3.1 3.1 Object-Oriented Design

The numerical_methods package is designed using an object-oriented approach. The main class in the package is the `ProblemSolver` class, which is designed to handle everything by analyzing the user's arguments. The class can handle both ODEs and PDEs, and it can solve a wide range of problems with different boundary conditions. The class has several numerical methods implemented, including Euler, Improved Euler, Midpoint, ODEStep, Runge-Kutta 4, Runge-Kutta 45, Crank-Nicolson, Explicit Euler, Finite Difference, Implicit Euler, and Method of Lines. The class automatically selects the most appropriate method based on the problem's characteristics, such as the type of problem (IVP or BVP) and the type of differential (ODE or PDE). The class can also solve problems with multiple solutions, such as boundary value problems, using the shooting method or the continuation method.

The other two significant classes are the `Solver` class and `Problem` class, these handle the numerical methods and the problem definitions respectively. The `Solver` class is a base class that is inherited by the `IVP` and `BVP` classes. The `Problem` class is a base class that not only defines the problem, but handles all the problem parameters and will raise errors accordingly.

3.1.1 Base Classes

3.1.1.1 Problem Class

- **It uses a keyword argument system to allow users to customize the solution of the problem.**

  The `Problem` class takes a number of keyword arguments, which can be used to customize the solution of the problem. For example, the user can input initial conditions, boundary conditions and parameters. Additionally, the problem equation keyword argument differs for ODEs and PDEs (`f` and `q` respectively) This design decision makes it easy for users to control the specifivisity of the problem whilst maintaining a simple interface.

- **Dynamic problem detection and extensible design**

  The `Problem` class can detect the type of problem being solved by analyzing the provided parameters, such as the number of function arguments, keyword names (e.g. `f` is used for ODE's and `q` is used for PDE's), and other indicators. This automated problem detection allows the `Problem` class to be flexible and easily extensible, as new problem types can be added by simply providing the appropriate parameters. It also allows the `Problem` class to provide informative error messages when the user provides invalid parameters and easily show them how to fix the problem. This removes the mystism of solving differential equations and makes it easier for users to get started with the package.

- **Comprehensive parameter validation and informative representation**

  The `_validate_parameters` method checks for the presence and data type of required parameters, and can also check for the presence of optional parameters and their data types. It can also compute any missing parameters such as the number of time dimensions, time step size, spatial dimensions and/or spatial step size.

  The `__repr__` method provides a nicely formatted table of the parameters, including equation and problem types, boundary and initial conditions, time and space attributes, and additional arguments. This informative representation facilitates debugging and parameter tuning (a feature i regularly used myself).

3.1.1.2 Solver Class

- **Flexibility and Robustness**

  To ensure that the `Solver` class is flexible and robust, I made several key design decisions. Firstly, I chose to make the `Solver` class an abstract base class, providing an interface for defining subclasses that can be instantiated and used to solve various types of mathematical problems, including IVPs and BVPs.

To achieve this flexibility, I stored all the attributes of the problem object in the `Solver` class, including the function `f` or `q`, initial or boundary conditions, time or spatial discretization parameters, and the method to solve the problem. I also ensured that the class could handle different types of input functions and arguments, including functions that take multiple arguments.

To maintain correctness and consistency of the input parameters, I included a series of checks and validations to ensure that the user provides the appropriate input. This included checking if the input problem object is of type `Problem` and if the chosen method to solve the problem is valid. Additionally, I made sure that the root-finding method, if applicable, is valid.

- **Multiple Solver Types**

  In order to provide flexibility in solving PDEs, I decided to include different types of PDE solvers, such as finite difference, finite element, and spectral methods. Furthermore, to handle continuation methods for solving BVPs, I implemented a method for handling continuation parameters.

  Through these design decisions, I aimed to create a powerful and versatile tool for solving a wide range of mathematical problems. The resulting `Solver` class provides flexibility and robustness while accommodating various input types and multiple solver types.

3.1.1.3 ProblemSolver Class

- **Composition to build up solutions to complex problems from simpler solutions.**

  The `ProblemSolver` class has two subclasses, IVP and BVP, which solve initial value problems and boundary value problems, respectively. When the class is instantiated, it creates an IVP or BVP object depending on the type of problem that is being solved. The class then delegates the actual solution of the problem to the IVP or BVP object. This design decision makes it easy to add new types of problems to the ProblemSolver class, since only a new subclass of IVP or BVP needs to be created.

  The use of composition makes it easy to add new types of problems to the `ProblemSolver` class. For example, if a new type of problem is added, only a new subclass of IVP or BVP needs to be created. The class does not need to be modified.

- **`solve` method handles IVPs and BVPs for ODEs and PDEs, continuation is optional.**

  The `ProblemSolver` class can solve initial value problems and boundary value problems for both ordinary differential equations and partial differential equations. The class can also solve problems with multiple solutions, such as boundary value problems, using the shooting method or the continuation method. This design decision makes it easy for users to solve a wide range of problems using a single class. Parameter continuation is optional, and can be easily enabled by passing keyword arguments to the `solve` method.

3.1.1.4 Solution Class

- **Design decisions for storing and representing numerical solutions**

  The `Solution` class is designed to provide a comprehensive solution for storing and representing numerical solutions for ODEs and PDEs. When initializing a `Solution` instance, arrays of time and solution values `t` and `y` are stored, along with solution values for PDEs in `u`. To handle continuation problems, the `params` parameter is used. The `problem_type` attribute is included to determine whether the solution is an ODE or PDE and whether it is a continuation problem. To provide a clear representation of the instance, the `__repr__` method returns a nicely formatted table including the problem type, time and solution values, and shapes of the arrays.

- **Design decisions for plotting numerical solutions**

  The `Solution` class also includes features for plotting numerical solutions using Plotly. The `plot` method returns a `Figure` object and allows users to customize plot dimensions, title, and axis labels. An exact solution can also be provided for comparison. For ODEs, the `phase_plot` parameter generates a phase plot in addition to the solution plot. This plot shows the solution values of one dependent variable against another dependent variable and can include the exact solution. Overall, the design decisions in the `Solution` class provide a comprehensive solution for storing and plotting numerical solutions for ODEs and PDEs, while also including additional features for phase plots and comparison with exact solutions.

3.1.2 Benefits of the design I structured my classes and objects in a way that reflects the real-world entities that the software is modeling. For example, the `Problem` class represents a mathematical problem, the `Solver` class represents a numerical method for solving a problem, the `ProblemSolver` class represents the tool for solving a problem, and the `Solution` class represents the numerical solution to a problem. This design decision makes the software easier to understand and use, since the classes and objects are familiar to users.

The key benefits of the design are as follows: - **Improved modularity**: The software is divided into smaller, self-contained modules, which makes it easier to understand and maintain. - **Improved reusability**: The software components can be reused in other projects, which saves time and effort. - **Improved scalability**: The software can be easily scaled up to handle larger problems. - **Improved performance**: The software can be optimized for performance by taking advantage of the modularization and reusability of the components.

3.1.3 Drawbacks of the design The main drawback of the design is that it is more complex than a procedural design. This makes it more difficult to understand and maintain. It can also be quite constrained as it is designed to guide the user through a specific workflow (i.e. specific keyword names required to choose problem type). This has the disadvantage of limiting the user's freedom to customize the solution of the problem. Furthermore, the types of functions that can be used to define the problem are limited to those that can be represented in a particular way. However, for the intended use case of the software, and with the functionality of error handling, I believe the design is appropriate as it provides a clear workflow for solving mathematical problems without requiring the user to have a deep understanding of the underlying mathematics.

3.2 3.2 Numerical Methods and Tools

This section describes the implementation of the numerical methods used in the software. The numerical methods include both explicit and implicit methods for solving ordinary differential equations (ODEs) and partial differential equations (PDEs), as well as shooting and finite difference methods for solving boundary value problems (BVPs). Some of the numerical methods are implemented using NumPy and SciPy functions for efficiency and accuracy, while others are implemented from scratch.

3.2.1 3.2.1 ODE Methods

3.2.1.1 ODE Step Methods (Euler, Improved Euler, Midpoint, RK4, RK45) The `OdeStep` base class was created to provide an interface for defining subclasses that can be used to solve ODEs. The `OdeStep` class includes two abstract methods:

```
- The `step` method, which takes the current time `t`, and returns the solution at the next ti
- The `_step_size_control` method, which takes the current time `t`, the current solution `y`,
```

The class was intended to easily build subclasses that implement different step methods for solving ODEs. The subclasses include `Euler`, `IEuler`, `Midpoint`, `RK4` and `RK45`. All of which easily adapt the `step` and `_step_size_control` methods to implement their respective step methods. On top of building the standard methods (e.g. `Euler`, `Midpoint`) I wanted to implement a method that common numerical method packages like SciPy use, which is the Runge-Kutta method. The `RK45` subclass implements the fourth-order Runge-Kutta-Fehlberg method. The `RK45` subclass also implements a step size control method that uses the fourth-order and fifth-order Runge-Kutta methods to estimate the local error and adapt the step size accordingly. This results in a more accurate solution than the other step methods.

3.2.1.2 ODE Discretization Method (shooting method) The `Shoot` class doesn't involve much complexity, however it does allow for a phase condition to be used (specified as a float), as well as a tolerance and maxiter for use with scipy.optimize.fsolve. I decided to stick to using my own implemented `IVP` solvers for the shooting method, as I wanted to keep the software as self-contained as possible. The `Shoot` class is used in the `IVP` class to solve boundary value problems using the shooting method if the `method` parameter is set to 'shoot'.

3.2.1.3 ODE Continuation Methods (natural, pseudo-arclength)

3.2.2 3.2.2 PDE Methods and Tools

3.2.2.1 Finite Difference Methods I chose to hold all the finite difference methods in one folder, this follows the same style as the ODE step methods. The `FiniteDifference` base class was created to provide an interface for defining subclasses that can be used to solve PDEs. The class includes two abstract methods:

```
- The `solve` method, which takes care of the main solving loop.
- The `setup_matrix` method, which takes care of setting up either the dense or sparse the mat
```

3.2.2.2 Root-finding Methods I implemented the Newton and method for root-finding. The Newton method is an iterative method that uses the derivative of a function to find the root. The `Newton` class doesn't follow any particular design apart from using finite difference approximations to calculate the newton step. The `Newton` class is used in the `FiniteDifference` class to solve PDEs using the Newton method if the `method` parameter is set to 'newton'.

3.2.2.3 Spatial Grids

- **Validation of input parameters**

  The `Grid` class validates the input parameters before creating the grid. This helps to ensure that the grid is created correctly and that it is suitable for the problem being solved. For example, the class checks to make sure that the domain is not empty, that the time step is not too large, and that the number of grid points is sufficient to resolve the problem. This validation helps to prevent errors and ensures that the grid is always created correctly.

- **Flexibility**

  The `Grid` class is flexible in the parameters it accepts. For example, the user can specify the domain, the number of grid points, the grid spacing, and the time step. This flexibility allows the user to create a grid that is suitable for the specific problem being solved. For example, if the problem is time-dependent, the user can specify a smaller time step to ensure that the solution is accurate.

- **Efficiency**

  The Grid class is designed to be efficient. The code is written in a way that minimizes the amount of computation required to create and manipulate grids. This efficiency is important for applications where the grid is created and manipulated many times, such as in time-dependent simulations.

3.3 3.3 Initial Value Problem (IVP) Solver

This subsection describes the implementation of the `IVP` solvers. The `IVP` class is a subclass of the `Solver` class and is used to solve initial value problems (IVPs) for ODEs and PDEs.

3.4 3.4 Boundary Value Problem (BVP) Solver

4 Section 4: Reflective Learning Log (3 pages)

4.1 Creating a Scientific Toolbox Package in Python

This experience has taught me a lot about the practical aspects of software engineering. I have learned how to write efficient and readable code, how to design software systems that are modular and extensible, and how to use version control to manage changes in code. The process of creating a scientific toolbox package has also taught me the importance of documentation and how to write clear and concise documentation for code. Overall, this experience has helped me to understand how software engineering concepts apply to real-world applications.

4.2 Short-Term Implications

The short-term implications of creating a scientific toolbox package are that I can use the package to solve scientific problems in various fields. For instance, I can use the package to analyze data in bioinformatics or to simulate solutions in continuum mathematics. Additionally, this unit has enhanced my coding skills and has prepared me for future programming assignments.

4.3 Long-Term Implications

The long-term implications of creating a scientific toolbox package in Python are that I can continue to improve my software engineering skills and contribute to the field of scientific computing.

The package I created can be further developed and extended to solve alternative problems with more sophisticated methods added on top. The experience has prepared me for future research opportunities in scientific computing, and I can use the knowledge and skills to delve into more complex tasks without feeling as daunted.

4.4 What Would I Have Done Differently?

I would have spent more time planning the software system's architecture and design before starting to write code. This would have helped me to avoid unnecessary code and to ensure that the software system is entirely scalable and decomposable. Furthermore, I would have sought more feedback from my peers and instructors to ensure that the package is of high quality and meets the project's requirements.

4.5 What Will I Do Differently in the Future?

In the future, I plan to apply the lessons learned from this unit to future programming assignments and research projects. I will strive to write efficient and readable code, design software systems that are modular and extensible, and use version control to manage changes in code. Additionally, I will seek feedback from my peers and instructors to ensure that my code meets high-quality standards. Overall, I plan to continue learning and growing as a software engineer in the field of scientific computing.