HARVEY MUDD COLLEGE

Joint Computer Science and Mathematics Clinic

Final Report for
Community of Ordinary Differential Equations Educators

ODEToolkit

April 29, 2011

Team Members
Beky Kotcon
Samantha Mesuro (Project Manager)
Daniel Rozenfeld
Anak Yodpinyanee

Advisors
Talithia Williams
Christopher Stone

Liaisons
Darryl Yong
Robert Borrelli

Abstract

Our sponsor, the Community of Ordinary Differential Equations Educators (CODEE), has been developing a software package called ODEToolkit that contains numerical ordinary differential equations (ODEs) solvers and aids in the teaching and learning of ODEs. Our contribution to this project has included making improvements to the structure of the software, implementing a numerical solver capable of solving stiff differential equations, and generally improving the software and its functionality.

Contents

List of Figures

Acknowledgments

We would like to thank our liaisons Professor Darryl Yong and Professor Robert Borrelli, our sponsor CODEE, and our advisors Professor Talithia Williams and Professor Christopher Stone, as well as Andres Perez and the rest of the 2008-2009 CODEE Clinic Team. Thank you to DruAnn Thomas, Claire Connelly, and Dr. Jeho Park for all of their assistance. We would also like to thank everyone who tested ODEToolkit, including Professor Adolph, Professor Levy, Professor Jacobsen, Professor Bernoff, and Professor Castro, for supporting the testing process. This project is supported by grant DUE-0717490 from the National Science Foundation.

Chapter 1

Introduction

1.1 Background

This clinic project is sponsored by CODEE, the Community of Ordinary Differential Equations Educators, which was funded by NSF/DUE (National Science Foundation/Division of Undergraduate Education) grants. Their mission is to "improve the teaching and learning of ordinary differential equations, primarily by encouraging broader use of modeling projects and computer experiments"(COD (2010)).

CODEE had the goal of developing software to aid in the education of ordinary differential equations that was free, accessible, and easy to use. In light of this goal and CODEE's mission statement, CODEE developed a software package in FORTRAN called ODE Architect that contains a library of models and robust numerical ordinary differential equation (ODE) solvers. However, this software is only usable on the Windows platform. CODEE wanted their program to be able to run on all platforms, to make it as accessible as possible.

To accomplish this, CODEE began to develop ODEToolkit, a newer version of ODE Architect written in Java. However, ODEToolkit was still not up to the standards of ODE Architect. In 2008-2009, a Harvey Mudd College Clinic team greatly improved the architecture of the code. This work was continued through the summer of 2009 by a Harvey Mudd student, Andres Perez. They succeeded in developing working code but some functionality was still lacking.

1.2 Problem Statement

For our clinic project, the primary goals are as follows:

1. Add a new numerical solver to allow ODEToolkit to accurately solve a larger class of ODEs.
2. Improve the software architecture.
3. Improve the program and its funtionality.

Additionally, we wrote documantation, added tutorials to the website, and recreated the old library of models from ODE Architect in the new ODEToolkit format.

1.3 Impact

ODEToolkit is intended to assist the teaching and learning of ordinary differential equations (ODEs). Students can use the program to explore ODEs, and instructors can create learning activities to help students understand ODEs. ODEToolkit is designed as a general tool for modeling ODEs, which will give students the experience of modeling and describing real-world events in scientific ways. Furthermore, since it is a free product, it will be especially beneficial to underprivileged schools.

1.4 Previous Work

1.4.1 08-09 clinic team

ODEToolkit was developed as a multi-platform version of the old ODE Architect software. Over the years, students hired by Professor Darryl Yong continued adding more functionality to ODEToolkit in an attempt to bring it up to the standards of ODE Architect. Unfortunately, they did not design the software with long-term development in mind. The lack of a clear design pattern eventually made it difficult to add new features without disrupting current ones. Therefore the task of the 08-09 clinic team was to restructure the architecture of the program to make it more modular. The 08-09 clinic team made significant progress in completing their task and greatly improved the architecture of ODEToolkit. However, they ran out of time before they were able to completely finish the restructuring of the software. They released ODEToolkit version 1.0.

1.4.2 Summer 2009 - Andres Perez

Andres Perez was a member of the 08-09 clinic team who continued working into the following summer to complete some of the work the team had been unable to finish. In addition to continuing the previous clinic teams' work, Andres Perez worked on adding functionality to ODEToolkit. One of his largest contributions over the summer was adding 3D capabilities. He released version 1.2 in July and then continued working for the rest of the summer. Thanks to a version control software called Subversion, his progress was saved in a repository at every stage of his work, so that nothing was lost even if he made large changes to the software. Near the beginning of our project, Andres Perez helped us to understand the different versions of the code, including version 1.2 and the code that he continued to work on afterwards. However, his most recent version of the software was not yet functional, so we decided to begin working from a functioning version somewhere between the 1.2 software release and his latest work in the repository.

Chapter 2

Plan of Action

2.1 Goals

The ultimate goal of ODEToolkit is to aid in the teaching and learning of ordinary differential equations. The long term goals of this year's clinic project were to deliver ODEToolkit in a functional state that is ready to be distributed freely via the CODEE website and to improve the functionality of the program by increasing the robustness and accuracy of its solvers.

2.2 Functionality Requirements

To help us focus our goals, our liaisons Professors Yong and Borrelli gave us a prioritized list of requirements for ODEToolkit to meet.

2.2.1 Minimal Requirements

ODEToolkit Minimal Function List

1. Open/save .ODE files to/from the local drive.
2. Be able to open .ODE files over the web, and launch ODEToolkit with a preloaded .ODE file via Java Web Start (JNLP files).
3. Be able to type in ODE system, initial conditions and solver parameters and have system parsed correctly (parser should have no less functionality than current system).
4. Be able to plot solution trajectories in x-t, y-t, or x-y phase space.

5. Be able to zoom in/out, pan viewing window. Be able to manually set viewing window.
6. Be able to turn on/off direction field (and for it to be accurately displayed).
7. Be able to print out current solution window.
8. Be able to view data points in some fashion for currently displayed solution trajectories.
9. Minimal software crashes and all UI elements are functional.
10. Old and new solvers can have their parameters tweaked, and have been tested and validated.

2.2.2 Stretch Goals

Nice to have:

1. Be able to have 3D functionality.
2. Be able to turn on logarithmic scale for one or both axes.
3. Be able to more flexibly choose which axes are being displayed on the graph.
4. Be able to change colors of solution trajectories.
5. Be able to annotate solution window by adding text.
6. Allow the user to work with multiple ODE systems and solution trajectories in a natural and flexible way (including how to get access to data points, solver parameters and ODE definitions of previous systems).
7. Be able to access an online library of model ODEs (so the library can be updated without updating the software).
8. Be able to parse higher-order differential equations.

2.3 Objectives

The primary objectives of our clinic project during the first semester were:

- Meet all requirements from the list of minimum requested functionality prepared by Professors Yong and Borrelli.
- Make the data structure more modular and enable it to store important information. Future developers should be able to quickly learn the code and add new features.
- Import a new numerical stiff ordinary differential equations solver into Java.
- Fix bugs and get the program into a state that does not crash.
- Provide thorough documentation. The code is currently not very well documented. In order to make it easier for future developers to work on this project, we should document the code, and provide clear notes describing how to continue developing ODEToolkit.

Our second-semester objectives were:

- Test the usability and functionality of the program in core mathematics classes at Harvey Mudd College once the software has reached a state with very few bugs, and then observe the students to see what could be improved about the user interface.
- Use a set of tests given to us by Professor Borrelli to check that the solvers had been implemented properly, and perform additional tests on the user interface to check that solutions and vector fields are being calculated and displayed properly.

We had some lower priority goals that we did not have time to implement:

- Implement any user-interface improvements found in user testing.
- Investigate other numerical solvers, and possibly implement them in ODEToolkit. This would involve evaluating different numerical methods based on their robustness and accuracy.
- Modify the existing parser with a new grammar so that ODEToolkit can handle a user input of higher order ordinary differential equations.

2.4 Original Deliverables

In light of these objectives, we outlined our deliverables in our Statement of Work. We planned to deliver:

- Fully documented source code
- Java documentation in web page format
- A test package for testing functionality and correctness of program, and results
- A tutorial for future developers
- Midyear and Final reports, presentations and a poster detailing our work

If time permitted we also planned to deliver:

- Code for a new parser
- Library of examples for classroom use
- An applet version of the software
- Documentation of user test results

2.5 Schedule

Early in the first semester we focused on doing research and other preparation for the project. We met with members of the 08-09 clinic team in order to help us familiarize ourselves with the code. During that time we were clarifying the expectations for the project with the liaisons. Midway through the first semester we received the finalized list of functionality requirements shown in Section 2.2. Around that time we chose the Rosenbrock solver for solving stiff differential equations. By the end of the first semester we coded the Rosenbrock solver in MATLAB and tested it for correctness. We also decided which version of the software to use as our starting point and began fixing bugs in order to get more familiar with the code.

Early in the second semester we decided on the architecture changes to be made and began implementing them. We integrated the Rosenbrock solver with the software and began testing it. By the middle of the second semester we added improvements to the solver, laid out our user test plan, and made it possible to save and load files. Now, at the end of the year, we have implemented many features, thoroughly tested the software and released it on the website.

Chapter 3

Software Progress

3.1 Original Software Architecture

To highlight the changes we have made in the architecture of the program, we will start by providing background information about the state of the software as we received it. It is worth noting that that latest version documented in this section is the version developed by Andres Perez during Summer 2009 after version 1.2 was released. Therefore, there are significant changes in the structure compared to version 1.0, which was released by the 08-09 clinic team. The information given in this section is chosen specifically to help readers understand the changes done by our team. Only information related to the changes is emphasized in this section.

1. **Launch**

   Launch is a package composed of only one class, ODEToolkit.java.

   ODEToolkit.java This class contains the main function that starts the program. This main function simply shows the splash screen and then begins a GUI. Launch will check to see if a web address to an ODE file is given. If yes, it will try to autolaunch (from the IO package) in order to open that ODE file and create an ODEWorkspace right away. Otherwise, the GUI will begin with a blank workspace.

2. **IO**

   IO is a package composed of classes related to reading and writing ODE files. It is composed of AutoLaunch.java, ODEFileHandler.java, and a subpackage IO.castor.

AutoLaunch.java This class contains a function LoadURL which tries to create a workspace from a given link to an ODE file, and if successful, removes the blank workspace initially created by the GUI. This process can be fixed so that the GUI tries to create a workspace from ODE file before creating an empty one first.

ODEFileHandler.java ODEFileHandler is used to read and write files in the ODE file format, a ODEToolkit-specific format which uses an XML structure to save ODEToolkit data in a file. In order to read an ODE file, the auto-generated code in IO.castor will create Java XML objects, which contain workspace information in tree format. Then the ODEFileHandler will convert these objects into Java objects (such as a workspace that is part of a running program that a user can interact with). The ODEFileHandler can then reverse this process to save the ODE file. This ODEFileHandler class was origionally not functioning correctly, but it was fixed after we completed refactorization so that it now contains the changes in ODE file formats. While this class contains a large amount of code and needs to be fixed manually, we have not found any better alternative for processing XML files.

IO.castor (package) This class contains code auto-generated by Castor that creates Java XML objects from reading the ODE save file, and writes a save file from Java XML objects. These Java XML objects must be converted to Java objects by ODEFileHandler.java in order to be used in the program. This package needs to be recreated whenever changes are made to the ODE save file format.

3. **Data Representation**

The Data Representation contains classes related to Workspace and PlotState.

UI The UI package contains information needed to interact with users. The classes in the UI generally correspond to objects shown to the user, such as a menu bar or a dialog box, as shown in Figure 3.1. We can divide the UI into 3 levels:

- **Program Level:** This level consists of components that all ODEWorkspaces share. The main objects in the Program level are the GUI and its components: the MenuBar, Status-

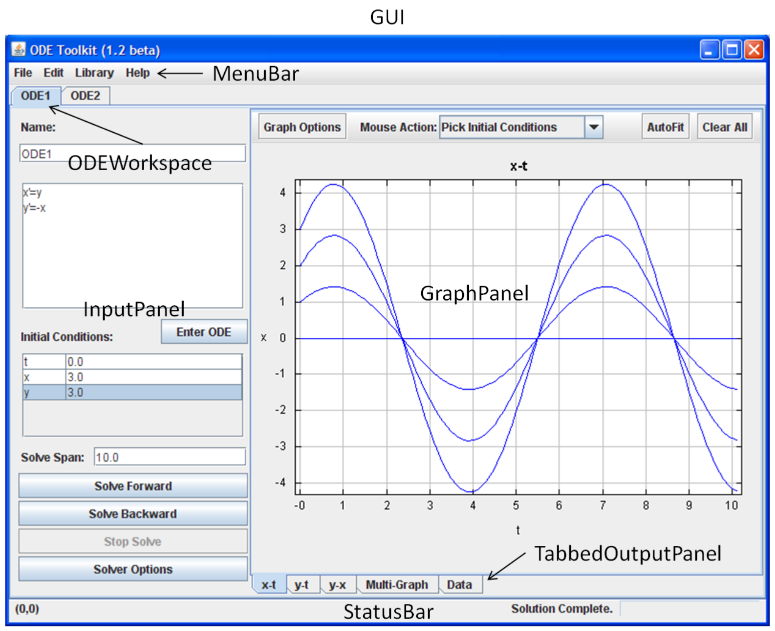


Figure 3.1 Components of the GUI.

Bar, several global variables shared by the whole program, and one or more ODEWorkspaces.

- **ODEWorkspace Level:** Each ODEWorkspace corresponds one user's project, which may include multiple ODEs. An ODEWorkspace corresponds to a single ODE save file. Currently, the class ODEWorkspace.java contains InputPanel, TabbedOutputPanel, and also WorkSpace data representation and Solvers. It accepts and processes all user input. Since this class is very complex, it is a main target of our refactorization. The classes in this level are in the outputpanels subpackage.
- **TabbedOutputPanel Level:** TabbedOutputPanels are the tabs that show user output in different formats such as GraphPanel, Graph3DPanel and DataPanel. The classes in this level are also in outputpanels subpackage.
  - GraphPanel is an abstract class, which is divided into different kinds of graphs, such as ComponentGraphPanel

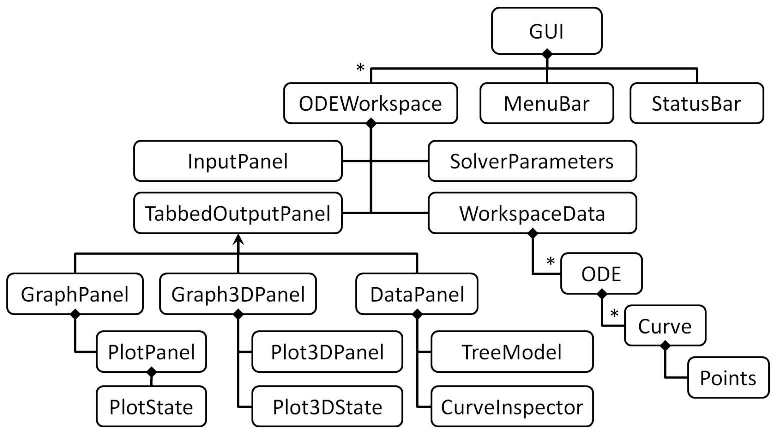


Figure 3.2 UML diagram of UI-related classes in the original architecture.

(plot between two dependent variables), PhaseGraphPanel (plot of a dependent variable with respect to time), and MultiGraphPanel (plot all dependent variables with respect to time). Each GraphPanel holds the PlotPanel, which contains a corresponding PlotState that stores how the plot is being displayed. GraphPanel also contains the toolbar and option dialogs that let the user customize the plot.

– Graph3DPanel holds the same type of information that GraphPanel does but is modified so that it supports 3D plots. In plotting 3D graphs, we use Plot3DPanel provided in the JMathPlot library, and thus we need to store the Plot3DState separately.
– DataPanel displays mathematical information about ODEs and their curves, which includes both data points and solver parameters information. It contains the TreeModel which helps display the ODE and curves in a tree format, and the CurveInspector which lets the user access the curve's numerical information. The option for editing solver parameters was not enabled, and after loading an ODE file, DataPanel did not display the list of ODEs and curves correctly. Currently the program cannot display both plots and data at the same time. DataPanel needed to be fixed along with the data represen-

tation and new ODE file format.

These classes are shown in the class diagram in Figure 3.2. The UI package also contains subpackage dialogs, which contain different dialog boxes or more specific panels, such as panel for adjusting the solver parameters for each solver.

Workspace A Workspace stores all mathematical information for a single ODEWorkspace. It contains all the ODEs in that ODE-Workspace. Each ODE then contains curves generated in that ODE. Each curve stores its points and other information specific to a curve, such as variable names, or solver parameters used to generate that curve. Data Representation also contains many helper classes, such as Curve, SolverSettings, Axis and Point-Comparer.

PlotState (package) PlotState contains information about how each plot is displayed on the screen, such as the range of each axis, or whether the direction field is turned on. Since there are different kinds of actions for different plots, there are classes specific to each kind of plot that inherits PlotState, including SinglePlot-State (for a normal plot between 2 variables), MultiPlotState (for a plot of the independent variable against other variables) and Plot3DState.

4. **Drawer**

   The Drawer package mainly draws the plots, by drawing background, curves, axis, tick marks and direction fields. The part of the code that draws 3D curves uses jmathplot, which is an external library, but still had a threading problem that caused the program to freeze when the user tried to plot the orbits of the ODE in the component graph.

5. **Parser**

   The Parser package is composed of auto-generated code used to parse the ODEs to a Java object that solvers can use. The parser currently does not support differential equations of higher than first order.

6. **Solver**

   The Solver package consists of all of the numerical solvers. It also contains a common interface for solvers (SolverInterface.java), the solver thread controller (Solver.java), and the parameters for solvers (Solver-Parameters.java). Adding a new solver requires adding a solver to this package, updating the solver parameters for this solver, and creating the dialog for configuring solver parameters.

7. **Util**

   The Util package simply contains useful classes. Some examples are listed below:

   - Listeners, such as PointClickedListener, which is an interface that notifies PlotPanel of the coordinates of the point that the user clicks.
   - TextFields, such as JDoubleTextField and JIntTextField, that allow the user to type specific types of numbers (real and integer, respectively).
   - Color-related classes, which manage the list of colors choices for the user, used to set colors for curves and direction fields.
   - SplashBitmap, which controls the splash screen that appears when the user runs the program.

3.2 Major and architectural changes

This section describes larger-scale modifications to the software, including the problem, solution, known issues, and possible future works for each modification. Bugs related to topics in this section will be included in the corresponding topics, while other minor bugs will be discussed in the next section.

3.2.1 Improve ODEWorkspace towards Model-View-Controller design paradigm

Before the 08-09 clinic team had worked on ODEToolkit, the software was mainly user interface (UI)-driven. The UI classes had access to data of other parts of the program, such as the solver and data representation, allowing them to directly manipulate related information according to user input. The team considered this to be inefficient design since the code is very complex, and changing some part of the code would generally affect other parts as well. The 08-09 clinic team tried to improve the program towards the model-view-controller (MVC) paradigm, which separates the program into three parts, each of them having a different role in the program and a different, highly-independent set of information. This makes the architecture more organized and more understandable. However, they did not finish the implementation, and the most important class that still needed to be refactored was ODEWorkspace. ODEWorkspace.java owned and manipulated the graphical component, the solvers, and the data representation.

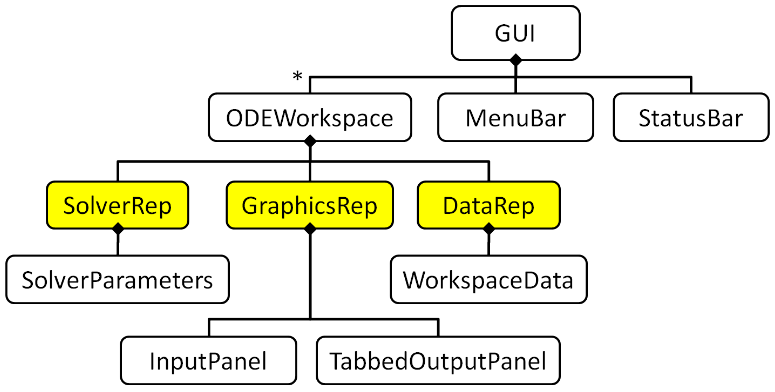


Figure 3.3 UML diagram of the improved ODEWorkspace structure.

We have successfully divided ODEWorkspace into the three classes as mentioned above, namely DataRep (model), GraphicsRep (view), and SolverRep (controller). These three new classes are shown in yellow in the class diagram in Figure 3.3. In the process, we tried to minimize references between each component by arranging the data stored in each part accordingly, and minimized direct reference of data from other parts of the program.

3.2.2 Improve WorkspaceData to fix variable problems and support new features

Properly manage the variables of the program

In the previous version of the software, the list of variables was stored as an ODEVarVector object by each ODE, but an overall list of variables from all ODEs was not stored anywhere. We also had an object ODEVar for each variable that contained the variable name and index, but this index was changed when the user input a new ODE. Therefore, the program did not respond properly when a new ODE with a different set of variables was entered into the program. For example, if we replaced an ODE using variables x, y and z, with a new ODE using variables a, b and c, then the old plots would show curves of the variables x, y and z as if they were the variables a, b, and c. The program also caused a null pointer exception when the new ODE had fewer variables as it tried to find a variable with higher index then we had. Lastly, some component plots were not available for a related pair of variables. For example, for an ODE with variables x, y and z, only the y-x plot was available for the user, not the z-y or z-x plots.

To solve this problem, we created a class called ODEVarKeeper that keeps the list of all of the variables that ever appeared in all ODEs in the workspace. We also assign each ODE variable a unique index in this list of ODEVarKeeper. The ODEVarKeeper is also responsible for creating new variables and helps with converting between variable names and indices. The index that each ODEVar keeps also corresponds to the list of variables in the ODEVarKeeper. Since the index is uniquely determined for each variable name, we can properly prevent problems arising from misusing the variables.

In the improved implementation, each ODE still stores an ODEVarVector object that contains the list of all variables in that ODE. However, with the indices with variables fixed, we can now iterate through all pairs of variables in this list to create all necessary curves between variables, while still plotting the curves from old ODEs correctly.

Allow users to change the visual properties of each individual curve

Previously, the visual properties of the curves were stored by the PlotPanel, and this property was applied to all curves in the PlotPanel. Thus, the curves in each PlotPanel, except for multi-graph, would have the same color regardless of whether they were from the same ODE or had the same solver parameters. This made it hard for users to distinguish curves from different sources, as suggested in Figure 3.4.

In order to allow the visual properties of each curve to be changed, we created a new class called VisualProperties, which stores the information about the curve color and whether it is shown (visible) on the plots. Each curve will then contain an instance of this VisualProperties object, and the plots will be referencing this object to specify how they should draw these curves. For example, if a curve is invisible, then the plot will not draw that curve, and if the auto-fit mode is on, invisible curves will also be ignored by the algorithm that determines the range of the plot. Figure 3.5 shows the screenshot of the program with this feature applied.

Note that in our design of VisualProperties, future developers should be

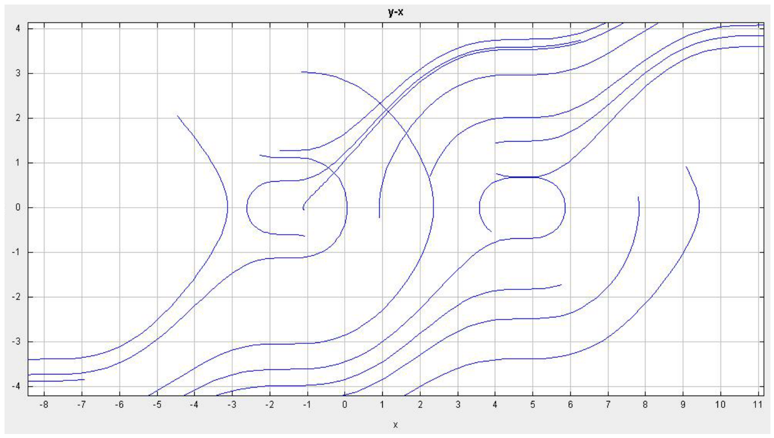


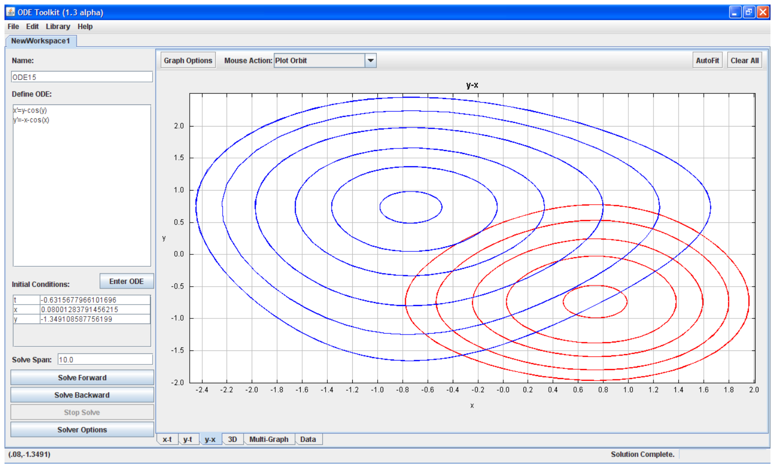
Figure 3.4 The screenshot above shows the plot of two different ODEs. It is impossible to distinguish the curves of these ODEs using the old software architecture.

able to extend this class very naturally to add more properties, both to this class and to the file format. For example, it would be useful to check which curve is selected, so that we can display that curve differently, allowing the user to distinguish the selected curve on the plots, which is a functionality provided in ODE Architect. Note also that the visual properties are not used to plot MultiGraph, because in MultiGraph, we need different curve colors for each variable, not for each curve object.

Store equilibrium points as objects owned by ODE class

Like curves, equilibrium points are objects specific to each ODE and should be displayed on appropriate plots. They can also have visual properties and be displayed through data tab as well. Originally, ODEToolkit treated equilibrium points as extra points to be drawn, stored by each plot. Thus the equilibrium points computed in one plot will not be shown in another. It also removed the equilibrium points when a different ODE was entered because the old plots were discarded. Furthermore, the implementation of the algorithm for finding equilibrium points also only allowed users to specify the initial guess for two of the variables, and so it can only compute equilibrium points on the (sub)system with two variables depending on each other.

To solve this problem, we created a class called Equilibrium, in which



Name: ODE15

Define ODE:

x'=y-cos(y)
y'=-x-cos(x)

Initial Conditions:

t	-0.6315677966101696
x	0.08001283791456215
y	-1.349108587756199

Solve Span: 10.0

(.08,-1.3491)

Solution Complete.

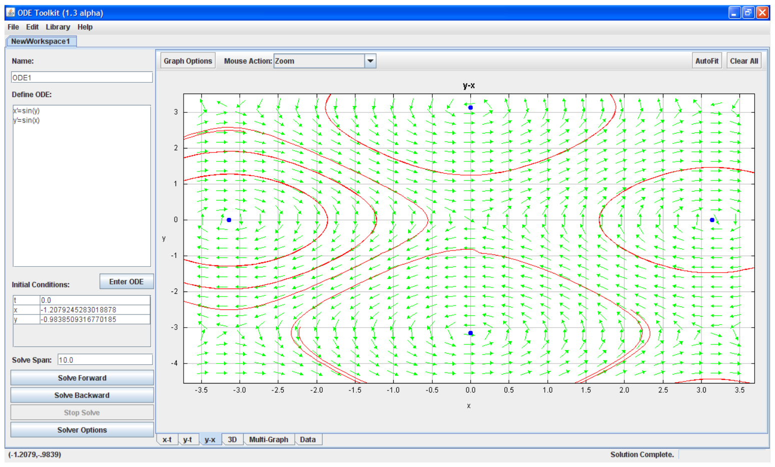
Figure 3.5 The screenshot above shows the plot of two different ODEs, using one color for each ODE.

we store an equilibrium point, along with a VisualProperties object, much like what we did with the Curve class. The ODE objects will then store a list of Equilibrium objects. Using this implementation, all plots will have access to the equilibrium points, thus all equilibrium points can be shown in all plots with different visual properties. Furthermore, equilibrium points will no longer be removed when a new ODE is inputted, and they can also be stored in the ODE save file. We also successfully generalized the code for obtaining the initial guess for the equilibrium points to any number of variables, making our software capable of computing equilibrium points of systems with more than 2 variables. Figure 3.6 shows the screenshot of the programs with equilibrium points and curves under different visual properties.

Figure 3.7 shows the new architecture of the WorkspaceData. The yellow boxes represent classes created to fix the ODE variables problem. The orange box shows the new Equilibrium Point class. The blue boxes show the visual properties added to both Curve and Equilibrium Point classes.

3.2.3 Add preliminary version of 3D feature

Imitating ODE Architect, we and previous developers of ODEToolkit have planned to add the 3D feature, by providing a 3D plot in the form of an additional tab along with other 2D plots and the Data Panel. Andres Perez



Name: ODE1

Define ODE:

x'=sin(y)
y'=sin(x)

Initial Conditions:

t	0.0
x	-1.2079245283018878
y	-0.9838509316770185

Solve Span: 10.0

Mouse Action: Zoom

(-1.2079,-.9839)

Solution Complete.

Figure 3.6 The screenshot above shows the plot of curves and equilibrium points in different colors.

started but did not finish implementing the 3D feature, and our clinic team decided to continue implementing this feature as a secondary goal after we had completed the minimum requirements. This 3D feature relies on the library JMathPlot, which provides its 3D plot canvas along with its toolbar, legend table, and mouse action for controlling the orientation of the plot. The original state of the 3D feature suffered from the variable indexing problem and caused a null pointer exception. Furthermore, when the user tried to plot orbits, the program regularly froze. 3D plots also lacked many features provided by 2D plots.

The indexing problem was fixed when we created the ODEVarKeeper. We investigated the program freezing problem and determined that it was a threading problem, where the Drawer class tried to draw both the 2D plots and 3D plots at the same time, resulting in each thread locking different information and freezing because of the deadlock. We successfully fixed this problem by carefully serializing the code so that these drawing processes are never done at the same time.

We also extended many features of 2D plots to 3D. For example, we can now plot curves and equilibrium points in 3D, using the VisualProperties of each curve and equilibrium point. The scaling and auto-fit features of the 3D plots are also done in the same way as 2D. We allowed changes in axis names, and added the plot title at the top of the plot. Additionally, unlike in 2D, we allow the user to choose the variable to display in each of

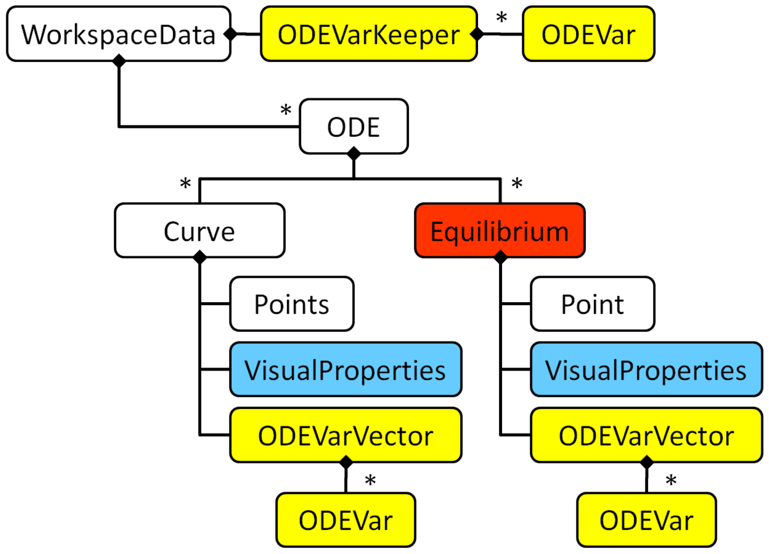


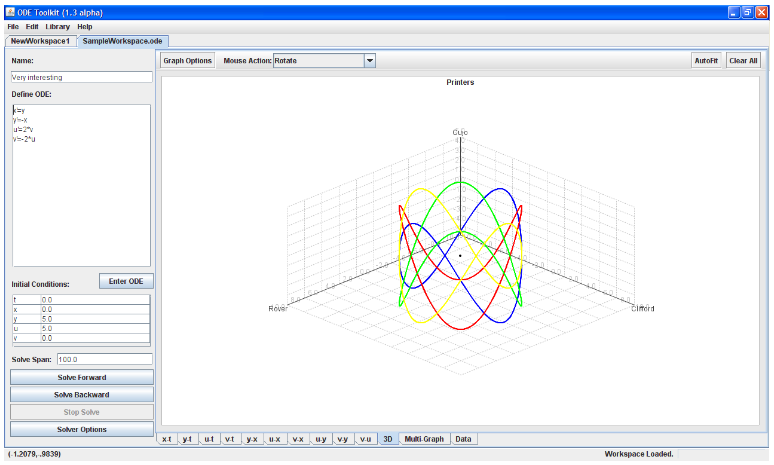
Figure 3.7 UML diagram of the improved software architecture for Workspace-Data.

the axis in order to prevent creating too many 3D plots at once, since in a system with n variables there can be as many as $\binom{n}{3}$ plots.

However, because we use JMathPlot, there are features of 2D that we cannot support in 3D. Firstly, we still cannot print or export 3D plots correctly due to the resizing problem. Secondly, the log-scale is not handled properly for curves and points with possibly negative values. Thirdly, the mouse actions for 3D do not match with those in 2D. For example, the right-click options for JMathPlot are reserved for other options, and we cannot display our options for 2D plots in 3D plots, and the users can only use the toolbar at the top of the plot. Lastly, the plot orientation cannot be accessed in the program, and thus we cannot store the orientation in the ODE save file.

3.2.4 Improve Data Tab

The Data Tab is one of the output panels that allow users to click at a specific ODE, Curve, or Equilibrium point, view its information, change its properties, or delete that object. This tab is included as the last tab, along with other output panels that provide plots between different variables.



Name: Very interesting

Define ODE:

```
x'=y
y'=-x
u'=2*v
v'=-2*u
```

Initial Conditions:

t	0.0
x	0.0
y	5.0
u	5.0
v	0.0

Solve Span: 100.0

Mouse Action: Rotate

(-1.2079,-.9839)

Workspace Loaded.

Figure 3.8 The screenshot above shows the 3D plots of curves and equilibrium point.

Originally, the data tab only allowed users to view the initial condition and data points of the curves. However, there were still other buttons in the data tab for functionality that was not implemented yet. The data tab was also not created correctly when the user loaded data from an ODE file.

We implemented much of the missing functionality for the data tab. For example, we implemented the solver dialog box that can now display the solver parameters that were used to numerically solve for the selected curve. We allow the user to delete the selected Curve object, and we include the visual properties information and allow users to change this information directly.

In addition to the CurveInspector which provides information about Curve, we added two new classes, EquilibriumInspector and ODEInspector, which work in the same way as the Curve inspector. Namely, the users can view information about the Equilibrium, such as points, or about the ODE, such as the system definition. They can change the visual properties of an equilibrium point, or every object in an ODE. The improved architecture of TabbedOutputPanel is shown in Figure 3.9.

We also fixed the implementation of the TreeModel, which is a Java object that helps provide the hierarchy structure of the ODE, Curve, and Equilibrium objects, which the user can select on the left side of the data tab. In the original software, the tree was updated every time that the user selected the data tab. Thus user's selection of an object would be reset, and

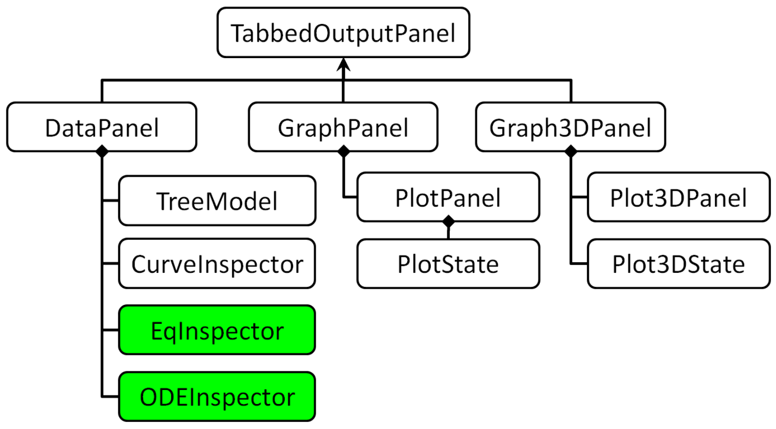


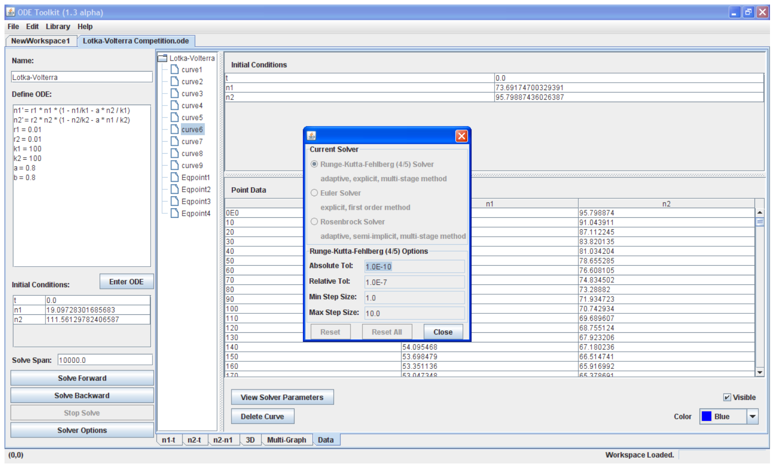
Figure 3.9 UML diagram of the improved software architecture for TabbedOutputPanel.

this process was quite inefficient. We now build the tree only once when we create or open a workspace, and only manipulate the trees as the user adds or deletes some object on the tree. We also fixed some minor problems, such as resizing the object label on the tree, and update the tree in real time as the objects are modified. The new screen shot of the data tab is shown in figure 3.10.

3.2.5 Improve ODE file format

We needed to improve the ODE file format for several reasons. When the software loads an ODE file, the data tab does not correctly show the list of ODEs and curves, even when loading an official ODE file given in the library. Moreover, the ODE file does not store information about the state of the program, such as which plot is being viewed, when the file is saved. Most importantly, in version 1.2 of the software, file saving and loading was completely non-functional.

After completing all improvements to the architecture of the program, we created a new XML file reader/writer via Castor. We then fixed the ODEFileHandler.java, which is the class that converts between the XML tree objects and the java objects used the program, so that it now works correctly. We also have improved the ODE file format in order to account for the new architecture of the software, and we store new options corresponding to the new features we implemented. The improvements include



Name: Lotka-Volterra

Define ODE:

```
n1'= r1 * n1 * (1 - n1/k1 - a * n2 / k1)
n2'= r2 * n2 * (1 - n2/k2 - a * n1 / k2)
r1 = 0.01
r2 = 0.01
k1 = 100
k2 = 100
a = 0.8
b = 0.8
```

Initial Conditions:

t	0.0
n1	19.09728301685683
n2	111.56129782406587

Solve Span: 10000.0

Initial Conditions

t	0.0
n1	73.69174700329391
n2	95.79887436026387

Current Solver

(●) Runge-Kutta-Fehlberg (4/5) Solver
adaptive, explicit, multi-stage method
( ) Euler Solver
explicit, first order method
( ) Rosenbrock Solver
adaptive, semi-implicit, multi-stage method

Runge-Kutta-Fehlberg (4/5) Options

Absolute Tol: 1.0E-10
Relative Tol: 1.0E-7
Min Step Size: 1.0
Max Step Size: 10.0

Point Data

	n1	n2
0E0		95.798874
10		91.043911
20		87.112245
30		83.820135
40		81.034204
50		78.655285
60		76.608105
70		74.834502
80		73.28882
90		71.934723
100		70.742934
110		69.689607
120		68.755124
130		67.923206
140	54.095468	67.180236
150	53.698479	66.514741
160	53.351136	65.916992
170	53.047348	65.378691

[✓] Visible

Color ■ Blue

Figure 3.10 Improved Data Tab interface and functionality: initial condition, data points and solver parameters of a curve can now be displayed

the following:

- We clearly distinguish between an ODE, which is the system of equations, and a Workspace, which is a space that user can use to work on multiple ODEs. The save file corresponds to a Workspace, not an ODE.
- We changed the naming convention of the Workspace, so that a newly created Workspace is named NewWorkspace followed by a number. Then the user is allowed to change the name of the workspace when saving an ODE file.
- A file cannot be opened on more than 2 Workspaces at the same time, and cannot be overwritten by another Workspace while it is opened.
- Equilibrium points are added to each ODE.
- Visual properties are added to each curve and equilibrium point.
- Indexing of curves and equilibrium points are stored, so that we can continue the same numbering the curves and equilibrium points after loading an ODE file.
- Information related to 3D plots, such as plot title and the extra axis, is added to the save file.

- All variables are consistently stored as names instead of the original indices system. This allows ODEVarKeeper to be reconstructed systematically.
- Each axis can store the variable name to display, which is required for 3D plots.
- Data points are stored as numbers instead of encoded as bytestrings, making the XML file more readable to the users.
- The code for ODEFileHandler is broken into smaller functions that are easy to read and debug, in contrast to the original version with more than 500 lines in each function.

Besides improving the file format, we have manually converted the files in the library provided by ODEToolkit into a new format as well, and these new files will be distributed along with the new version of ODEToolkit.

3.3 Minor fixes

3.3.1 Fixing the direction field

The direction field was rarely displayed correctly even though it gave a reasonably close trend in general. We discovered that the problem arises because, while the derivative of the ODE is correctly calculated, the size of the screen and the increment of value for each axis (per pixel) are not taken into account. Moreover, the original code does not support the direction field feature for log-scale plots, where the increment of value per pixel on each axis is not a constant. This problem has existed since version 1.0.

We have fixed this problem by taking into account the size of the screen and the increment of value per pixel. We also implemented functions for log-scale that compute the increments, making the direction field display correctly in log-scale as well.

3.3.2 Improving the viewing range specification

In the original state of the software, there are many bugs related to computing the viewing range of each plot. These problems are mostly generated from errors in mathematical calculation or the logical flow of the program, and can be fixed without structural changes.

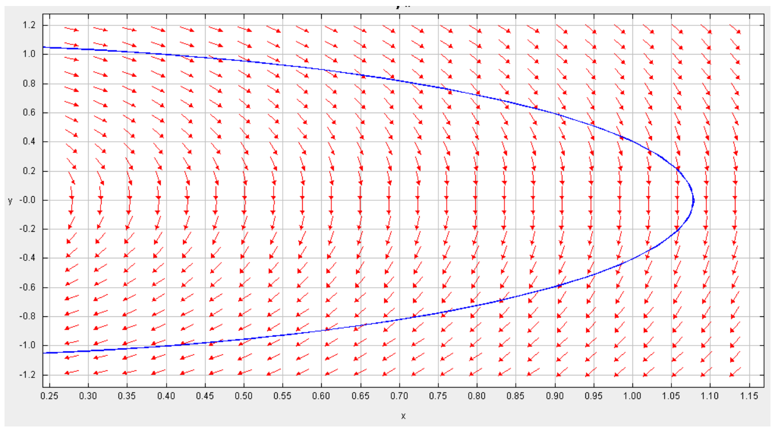


Figure 3.11 Direction field problem in the original software.

Fix auto-fit

Auto-fit is the feature that automatically adjusts the plot range to include only parts of the curves that are meaningful. We found that in rare cases when auto-fit is turned on, after solving for a curve, the program would freeze. This was caused by the function that tries to calculate the appropriate range for viewing the curve so that the plot includes all "important points" (such as inflection points), while at the same time, does not zoom out too far to show curves that do not have any interesting changes (such as going to infinity). In this function, there was a bug that causes the minimum and maximum range on x-axis to be reversed, causing the increment in x-value as we move along the axis to the right to be negative. This value is used to determine the appropriate ticks on x-axis whose formula involves a log function that causes the program to freeze when negative increment is given. This problem has existed since version 1.0. We have fixed this problem by making sure that all x-values for all important points are calculated and returned in the right order.

We then adjusted the algorithm for finding important points, so that for cases where there are few important points, these points are displayed closer to the middle of the screen. For example, if a single horizontal curve is present, it will be displayed at the middle of the plot under auto-fit mode. This problem occured in the original software because the initial range that contains important points is set to between -1 and 1, and this range cannot be removed in case of few important points. This range is set to prevent the reversed minimum-maximum range as mentioned above, even though it still does not work properly. So by fixing the reversed range as mentioned above, we can now set empty initial range, so that we can now display important points in the middle of the plot.

Auto-fit was also generally not performed at the correct occasion. For example, the plots were not rescaled when a new ODE is inputted by the user, or a workspace is loaded, even though in these cases, Auto-fit may be turned on either by the user or by default. This problem can be fixed by setting a boolean flag in the each plot that an auto-scaling is needed in the right situation. This boolean flag will then be referred to as the plot is selected, and rescaling the plot only when required. This technique allows the plots to be scaled at the right time, while does not require much overhead time to rescale all plots, which could be large when many curves with large number of points are present.

Limit zoom-in range

Previously, if the user zoomed in too much into the plot, the software was not able to handle the precision, which then caused a mathematical runtime error and freeze. This freeze occurs because the distance between tick marks on some axis was rounded down to 0, and thus the software kept generating tick marks which caused an infinite loop. This problem was easily fixed by making sure that users are not allowed to zoom in too much. Since this issue is dependent on the screen size that the software is run, we experimentally determine the minimum viewing range on each axis to be 10^{-10}, which should prevent the problem from occuring, while displaying accurate result for most educational ODE systems.

3.3.3 Improve memory usage

Reduce memory usage in solving algorithm with adaptive step size

Even though only the minimum and maximum step sizes are needed to compute step size in an adaptive-size solver, the original software requires the user to input the resolution for solving, so that the solver can use this resolution to compute the amount of memory needed. The solver then solves for the curve. If the allocated memory is enough, then it returns the unused memory to the heap. At the same time, the code does not handle the case where the allocated memory is not enough, and will throw an array index out of bound exception. While we could fix this problem by using the reciprocal of the minimum step size as the bound on the resolution instead, this bound could be very inefficient because the step size could be arbitrarily small.

In order to solve this problem, we made the solver only allocate more memory when needed. We cannot afford to allocate memory for a point every time because this requires too much overhead memory, and this format is not supported by plot functions. So, we created a PointsManager class that will allocate only one point at first. Then as the solver add points, the PointsManager will double its size to add the new point. The advantage of this method is that it does not require to know the amount of needed memory in advance, while the amount of memory used will be asymptotically the same as the actual needed space. Note that, since we do not rely on the resolution, we remove this field from the option dialog.

Fix the bug that doubles the solve span

ODEToolkit has a feature that when the user tries to solve for curves with the same solver parameters (including the initial conditions), it adds to the solve span of the curve by the original amount of solve span. However, we found that if the user clicked solve too quickly, then the program would suddenly run out of memory.

The original software implements this feature by keeping track of the solve span used to solve for a curve, both in forward and backward directions. Then if we obtain the same solver parameters as the previous solve, we will extend the solve span before solving. However, it does not check whether the solver is available; that is, whether the solver is in the process of solving another curve. In this case, the solver will not solve for a new curve. However, the extended solve span is not rest back to original. Thus if the user clicks solve too quickly, then the solve span will be doubled. Once the solver is available, the solve span for the solve will then become very large, causing the program to run out of memory.

To solve this problem, we created a new variable to keep track of whether the solver is available. We can use this variable to check whether the solver is available, and if the solver is not available, we then can reset the solver parameters accordingly. Note that we do not report this back to the users because the user can always see that the solver is unavailable, because the progress bar will be running and the solve buttons will be disabled.

3.3.4 Fix bugs related to updating solver parameters

Make sure that the solver parameters are updated as soon as the user inputs them

The solver options are updated by the program along with the new initial conditions when the user clicks on the plot (and not by inputting the initial condition directly into the box on the left). This confuses the user because the options are already changed, but the user has no proper way of learning which solver is being used, as debugging messages are only shown to the developer. To fix this problem, we make sure that the ODEWorkspace updates the status of its solver parameters before solving starts.

Make sure that the solve span is not reset before we update

Unlike parameters specific to individual solvers, the solve span is read when we need to solve, not right after the user inputs a new solve span. Previously, when the user input new initial conditions by clicking at the plot, as mentioned, the software would update the solver parameters along with the new initial conditions. Then, it reset the interface to reflect the updated solver parameters. However, since the solve span had not been updated yet, it would be reset to the old solve span stored in the solver parameters. To fix this problem, we simply created a separate function to update only the initial condition and not other solver parameters including the solve span.

3.3.5 Fix UI-related bugs in GraphPanel and PlotPanel

GraphPanels mouse actions drop-down menu gives NullPointerException in Windows and Linux only

In Windows and Linux, when the user clicked the drop-down menu at the top of the plot, sometimes the menu would not show up and a null pointer exception occurred. This was caused by the null "viewing window" object being passed around when the program computed the locations to draw the menu. To fix this problem, we simply added more code to handle this condition.

Fix available menus for different PlotPanel

There are many types of PlotPanel, including PhaseGraphPanel, ComponentGraphPanel, and MultiGraphPanel. Each of these plots should allow different options. However, the original software does not display available options correctly. We fixed this problem by using the following rules for each possible option:

- A direction field can be displayed only on the PhaseGraphPanel and ComponentGraphPanel if and only if the two variables displayed on that plot are not dependent on any variables except for themselves and each other. It cannot be displayed on MultiGraphPanel.
- Plot orbits and Find equilibrium point options are allowed only on the PhaseGraphPanel, where the independent variable t is not involved.

Make sure that graph options in multiple places are synchronized

For convenience, ODEToolkit allows the user to select graph options from both the options menu above the plot and by using the right-click menu. However, updating options in one menu (such as toggling whether direction fields are on) would not change the status in the other menu. We solved this problem by finding the method that changes the status of the menu when the other menu is modified. The following parts of the menus are fixed.

- For the direction field, we made sure that the toolbar menu and right-click menu are changed together by creating a centralized function that changes both menus together, and do not allow each menu to be changed separately.
- For the auto-fit option, we also created a centralized function that changes both menus. However, for auto-fit option, this option can also be changed in the scale dialog, and thus this function also includes the scale dialog.

Other parts, such as log-scale option and mouse action are already implemented correctly in the original software.

Use repeated colors for MultiGraphPanel with many variables

While the user can assign visual properties to curves and equilibrium points, we do not use these visual properties in MultiGraphPanel because the color for MultiGraphPanel depends on the variable. In the original software, colors are selected from a predefined list of colors. Thus an error will occur if the number of variables exceeds the number of colors. We fixed this problem by reusing the colors, and increase the number of colors for the plot. We also uses the indices of the ODEVar to select color, so that ODEs with a different set of variables are not plotted with same colors.

Chapter 4

Remaining Defects

4.1 Known bugs

4.1.1 Behaviors of 3D Plot

Since we use JMathPlot library to create 3D plots, we cannot fully control how the plot is computed or updated. The plot, at this current stage of software, resets to its autofit bound after many user actions, such as panning or clicking at Graph Options button. This may be fixed by hacking into the library, or use a different library for 3D plots.

4.1.2 Inaccurate equilibrium point finder

Our user test suggested that when using the Find equilibrium point feature multiple times, the user can still zoom into the plot and visibly see differences in equilbrium point positions, even though those points, theoretically, should be the same point. This happens particularly frequently in systems where the derivatives at positions close to the equilibrium point decrease exponentially, such as the Lotka-Volterra Competition Model, which was used in the user testing.

4.1.3 Repainting plots

- When the user clicks the Graph Options menu, the whole plot briefly shrinks to the size of the pop-up menu, before returning to its normal display and showing the menu.
- For the scale dialog box, if the dialog is invoked via the toolbar at the top of the plot, the axis scale will be changed as the user edits it.

On the other hand, if it is invoked from the right-click menu, the axis scale will be changed after the user clicks OK.

- While computing curves for plot orbits, if the direction field is on, then the direction shown becomes incorrect. However, when the solving is done, the direction field turns back to normal.

4.2 Possible improvements for future developers

This section includes the list of features or possible improvements that could be added in the future. Some of them are already implemented, but contain problems and will be disabled until the problems are fixed.

4.2.1 Implement deleting and retrieving ODE

Currently, in the data tab, the users can access the old ODEs that they inputted. However, they cannot delete those ODEs or change the environment of the program back to old ODEs. This will require careful implementation as there are many components of the program that depend on the ODE. It also requires more work in the interface design, because it is possible for the Workspace to end up with no ODEs, much like a new Workspace that has just been created.

4.2.2 Add functionalities of 2D plots to 3D plot

As mentioned, because we use JMathPlot, we cannot specify the features of the 3D plot, including the mouse action, printing and exporting, log-scale, and storing the orientation of the plot. To achieve this, the code for JMathPlot may need to be overwritten in order to support these new features. We can also consider alternatives to JMathPlot as well.

4.2.3 Allow selecting and identifying a selected curve

In ODEArchitect, the user has a extra mouse option to click on the plot and select a specific curve. This feature is useful because the data points shown in data tab are generally not enough to determine the corresponding curve, as there are many overlaps between curves. We can consider adding another field to the visual properties so that it keeps track of whether a curve is selected, and plot this curve differently. However, selecting a curve from a plot requires an algorithm to find a curve closest to the selected point on the screen. Since the screen is affected by the range of the plots being displayed, then this algorithm could be complicated and hard to implement.

4.2.4 Add annotations to the plots or ODEs

This feature would allow users to put information about the ODEs onto the screen, which could also be included when printing the curves. For example, instructors may want to include a problem statement with the ODE and plots for students. Implementing this feature will involve modifications to the user interface as well.

4.2.5 Parse higher-order ODEs

The software currently only accepts systems of first-order ODEs, and we still do not have a thorough idea of how the software should interpret higher-order ODEs. By allowing users to input higher-order ODEs, users may create contradicting equations that make the ODEs numerically unsolvable. Furthermore, we need to develop an interface that allows users to input multiple initial values for one variable; for example, we need to know both y and y' at some t to solve the differential equation $y'' = -y$. We also need to rewrite the input parsing grammar to correctly interpret a higher order equation.

Chapter 5

Solver Progress

5.1 Original State of Solvers

5.1.1 Background Information on Numerical Solvers

There are many different types of numerical solvers. In general these solvers discretize the domain of the independent variable (refered to as t_i here) and approximate the value of a function y at each of those t_i values. The notation y_i will be used to denote the approximation of the value of y at t_i. Here we go over some of the more important types of numerical solvers and discuss the benefits and costs associated with each.

Explicit vs implicit

- In explicit solvers, the approximation for the next iteration does not appear in the difference equations used for calculating that approximation, which makes solving for the next approximation straightforward. An example of an explicit difference equation is shown below. $y_{i+1} = f(y_i)$, where f is some function of y_i.

- In implicit solvers, the approximation for the next iteration is included in the expression for that approximation. This means that additional work must be done in order to solve for the approximation, beyond simply plugging in values. While implicit solvers are more accurate, they can sometimes be difficult or even impossible to run. This occurs because some implicit equations cannot be solved explicitly for the approximation being calculated. This problem can be overcome by using a Predictor Corrector Method that combines explicit and

implicit methods. An example of an implicit difference equation is shown below.
$y_{i+1} = f(y_{i+1}, y_i)$, where f is some function of y_i and y_{i+1}.

Fixed step size vs adaptive step size

In some methods the size of the intervals in the discretization of x is fixed. Other methods however, may change the size of the intervals depending on how large the calculated error was at the last time step. The Runge-Kutta-Fehlberg solver is an example of an adaptive step size method.

Stiff differential equations

Another distinction between numerical solvers of ordinary differential equations is whether or not they can approximate the solutions of stiff differential equations. Stiff differential equations have the property that the value of the derivatives of their solutions can change rapidly over time. This results in extremely large errors when the derivative at one point is used to approximate the derivative of the solution over a period of time. Stiff differential equations are often of the form $y' = ky + f(t)$, where k is a constant large in absolute value. Solvers that are better suited to handle stiff differential equations have what is called a larger region of absolute stability. When a numerical method is applied to the test equation $y' = ky$, the difference equation can be reduced to the following form, $y_{i+1} = f(hk) * y_i$, where h is the step size (the size of the discretized intervals of the independent variable). This implies that $y_n = f(hk)^n * y_0$. The region of absolute stability is then the region in the complex plane where the absolute value of $f(z)$ is less than one, where z is a complex number and has taken the place of hk. As you can see, the larger the region of absolute stability, the larger h can be before the solution grows rapidly and inaccurately. Of the solvers that were included in ODE Architect, the CVODE and LSODE solvers have a larger region of stability and are able to solve stiff differential equations. We will discuss these solvers further in the following section.

5.1.2 Original State of the Numerical Solvers in ODEToolkit

In ODE Architect, the predecessor to ODEToolkit, there were four numerical solvers. These four solvers used Euler's method, the Runge-Kutta-Fehlberg method, the Livermore Solver of Ordinary Differential Equations

(LSODE) and a C-language Variable-coefficient Ordinary Differential Equations (CVODE) solver. The way that the CVODE and LSODE solvers were implemented in ODEToolkit 1.2 only allowed them to work on the Windows operating system. The Runge-Kutta-Fehlberg solver and the Euler solver are written in Java and so are already platform independent. However, the CVODE and LSODE solvers were important to ODE Architect because they can numerically approximate the solutions of stiff ordinary differential equations whereas the other two solvers cannot.

5.2 Progress on Solvers

5.2.1 Our Work on the Numerical Solvers

We needed to add a solver to ODEToolkit, written in Java so that it easily works on all platforms, that would be capable of accurately approximating the solutions of stiff ordinary differential equations. We had to choose between converting either the CVODE or LSODE solvers from Fortran into Java or finding a new solver with a large region of absolute stability. We also had to decide whether to machine translate a solver from another language or to reimplement it by ourselves. If we chose to machine translate either the CVODE or LSODE solvers from Fortran into Java, the resulting Java code would have most likely been unorganized and difficult to understand. We would have had to treat the resulting code as a black box and only consider what it outputs rather than why. Although a machine translator could have saved us considerable time, its disadvantages outweighed its merits so we decided against it.

5.2.2 Investigating CVODE, LSODE, and Rosenbrock methods

We researched CVODE, LSODE and a class of methods known as the Rosenbrock methods, in order to evaluate whether we would be able to implement them ourselves and to decide which method we would like to implement. CVODE and LSODE are the two solvers that were in the original version of ODEToolkit, and they have many similarities. They are both able to solve stiff and non-stiff equations and they use combinations of similar numerical solvers based on difference equations. To choose between them we also had to consider their differences.

CVODE and LSODE

The two numerical ODE solvers in each of these packages use Adams-Moulton and Backward Differentiation methods. Adams-Moulton is an implicit method and it is used in a variable order, variable step form in both CVODE and LSODE. It is used for solving non-stiff equations. Backward Differentiation is also an implicit method but it depends only on the previous approximations and the value of $f(y)$ at the current approximation. It does not take into account the function evaluated at any of the previous approximations. It is especially used to solve stiff equations.

Since both of these methods are implicit, additional work needs to be done in order to solve for the desired approximation; one cannot obtain the approximation by simply plugging in values from previous iterations as in explicit methods. In order to do this, CVODE can use either of two iteration methods: functional iteration or Newton iteration. Functional iteration is primarily used for non-stiff problems while Newton iteration can be used in either case. Newton's method requires solving linear systems and CVODE contains a few different options for how to do so based on the form of the Jacobian matrix

$$\begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

(Cohen and Hindmarsh (1996)).

Similarly, LSODE can use any of three iteration methods: functional iteration, Newton-Raphson (NR) iteration, or Jacobi-Newton (JN) iteration. As pointed out in Radhakrishnan and Hindmarsh (1993), for convergence functional iteration requires $h_n\beta_0 L < 1$, where h_n is the n^{th} step size, L is the Lipschitz constant, and β_0 is the coefficient of the first corrector term in the approximation of the solution. In other words, β_0 is the coefficient for the first additional term added to what otherwise would be a basic numerical method similar to Euler's method. Since stiff equations are characterized by having large Lipschitz constants, convergence would require a very small step size, demonstrating why functional iteration is not useful for stiff equations. Newton-Raphson (NR) can use larger step sizes and converges quadratically, which is faster than functional iteration. It involves solving for the zero of an equation, which requires computing the Jacobian matrix and other matrix computations. Jacobi-Newton iterations are a variation of NR iteration that ignore all entries of the Jacobian matrix off the main diagonal. This reduces the amount of computation required to perform the iterations but also slows down the rate of convergence (Radhakrishnan and Hindmarsh (1993)). Clearly there are tradeoffs with each of these methods in terms of speed and computational efficiency. That is why it is beneficial that LSODE can use any one of the three. From this research, it seemed that LSODE is a little more flexible than CVODE. However, it seemed that the differences between them were fairly negligible.

Rosenbrock methods

The other methods we researched were the Rosenbrock methods which we discovered on a webpage that compares several different methods for solving stiff ODEs (Web (2010)). The bulk of our information on these methods came from $NumericalRecipes$ (Press et al. (2007)). In addition to describing this method, the textbook also provided some sample code to help implement it.

These methods are different from LSODE and CVODE because they are not packages of multiple solvers and do not require iteration methods. The Rosenbrock methods are semi-implicit: in order to solve for y_{n+1}, the linear approximation of $f(y_{n+1})$ is used. The Rosenbrock methods are generalizations of the Runge-Kutta method. The general formula of an s-stage Rosenbrock method is shown below:

$$\mathbf{y}(t_0 + h) = \mathbf{y}_0 + \sum_{i=1}^{s} b_i \mathbf{k}_i \tag{5.1}$$

where h is the step size, the b_i are constant parameters, and the k_i can be solved for by solving the following linear equations:

$$(\mathbf{1} - \gamma h \mathbf{f}') \cdot \mathbf{k}_i = h * f(\mathbf{y}_0 + \sum_{j=1}^{i-1} \alpha_{ij} * \mathbf{k}_j) + h * \mathbf{f}' \cdot \sum_{j=1}^{i-1} \gamma_{ij} * \mathbf{k}_j \tag{5.2}$$

where the γ_{ij}, the α_{ij}, and γ are constant parameters, $\mathbf{1}$ is a ones matrix, and $\mathbf{f}'$ is the Jacobian matrix.

Using different sets of parameters, many different Rosenbrock methods can be derived. The parameters that we used in our coding were supplied by a web resource of $NumericalRecipes$ (Press et al. (2007)) and they were described as the parameters for the Rosenbrock Stiffly Stable Method.

From the information gathered during research, we decided to implement a Rosenbrock method rather than the LSODE or CVODE solvers. We made this decision primarily because the Rosenbrock Method is far simpler to implement than the other solvers. There are a few downsides to the

Rosenbrock method however. First, the Rosenbrock Method is not as accurate as CVODE or LSODE, but since ODEToolkit is meant for educational purposes, the Rosecnbrock method is sufficiently accurate. Another large drawback is that unlike LSODE and CVODE, this method is not made to adapt for stiff and non-stiff systems. It is a method specifically for solving stiff systems and is inefficient for non-stiff systems compared to other methods. However, due to our desire to have a solver that could handle stiff systems we decided to implement a Rosenbrock method.

5.2.3 Stages of Implementing Rosenbrock Method

We started by implementing the Rosenbrock Stiffly Stable method in MATLAB because we were more familiar with it than Java and because we could use many of its built-in tools to assist in our early coding. Throughout this coding process we used the example code in $NumericalRecipes$ as a guide. We performed the implementation in stages, increasing the complexity of ODEs that the solver could handle, from single autonomous ODEs onto systems of non-autonomous ODEs. For the first round of these stages we let df/dy and df/dt be inputs rather than trying to calculate them within the code. At each stage we tested the solver on stiff and non-stiff equations whose analystical solutions could be computed for comparison, only moving on when the numerical solutions were accurate.

In the second round of implementation we needed a way to find the derivatives of f within the code rather than taking them as inputs. We looked at two different methods for this, symbolic differentiation and finite difference approximations. We decided to use finite difference approximations as the symbolic differentiation would have been much harder to implement. Once we'd implemented finaite difference methods in MATLAB and tested it, we were ready to translate the code to Java.

Solving systems of ODEs required us to implement a solver of matrix equations. For this we used MATLAB's built-in functionality, which was reasonable because Java's math libraries have equvialent functionality which we used when translating the code. We also tried to code some lower-level functions (such as matrix inversion) in Java, in order to familiarize ourselves with Java by working on simple problems. Additionally, we started exploring the solvers that already exist in ODEToolkit, both to additionally familiarize ourselves with Java and to enable us to model this Rosenbrock method after the existing solvers in ODEToolkit.

Once we had familiarized ourselves with the solvers that were already in ODEToolkit, specifically the Euler solver and the Runge-Kutta-Fehlberg solver, we were able to translate our Rosenbrock solver from MATLAB into Java. We used the code for the Runge-Kutta-Fehlberg solver as a guide since it is a similar multi-stage method. After the class file was written (RBSolver.java) it was time to integrate it into the software. We made the necessary changes of adding this solver to the list of solvers in the user interface.

5.2.4 Testing the Rosenbrock Solver

After the solver was integrated into the software we performed a series of tests to check it for functionality and accuracy. We started with some informal tests of both stiff and non-stiff DEs to get an idea of whether it worked or not. Once we saw that it was working relatively well we began to run more rigorous tests, using the DEs that we had used to test the MATLAB version of the solver. The specific tests are in Appendix A.1. For these tests we compared the output data points to values calculated from the analytical solution. In that way we were able to calculate the error in our solution to see whether it was reasonable and, in some cases, compare the performance to the Runge-Kutta-Fehlberg solver.

Our liaison, Professor Borrelli helped us in this testing process. He had prepared a test suite (shown in Appendix A.2) in order to test solvers in the past which checked the accuracy both of the data output and graphical output of the solution. He performed each of the tests on the Rosenbrock solver and gave us the results and feedback so that we could fix any problems with the software. Through this we realized that the solver was not able to correctly handle DEs whose solutions escaped to infinity in finite time. This led us to investigate ways that numerical solvers detect singularities in the solutions that they produce, in order to fix this problem with the Rosenbrock solver.

5.2.5 Solving the Singularity Problem

We started by looking at the Runge-Kutta-Fehlberg solver that already existed in ODEToolkit. We found that because the step size is adaptive in that method, the Runge-Kutta-Fehlberg solver did not have the same problems with solutions that escaped to infinity as the Rosenbrock solver. As the solution grows toward infinity, a smaller step size is required in order to calculate the solution within the error tolerance, and as a result the minimum step size is eventually reached and the solver stops.

Further research revealed that many other numerical solvers work in the same way. Most commonly used numerical methods include some way of adapting the step size in order to improve accuracy and efficiency. This feature allows them to more easily handle ODEs whose solutions have singularities by enforcing some minimum step size or maximum number of iterations. We found this to be the case for numerical solvers in applications such as Maple and Mathematica. There were not many other alternatives in the literature, and implementing a method for adapting the step size in the Rosenbrock solver seemed to be the most feasible option.

Thus we looked into ways of adapting the step size within the Rosenbrock method. Numerical Recipes contained a method for approximating the error in the solution approximation. So we first tried mimicking the Runge-Kutta-Fehlberg solver's algorithm for adjusting the step size, using this error approximation from Numerical Recipes. Upon testing this new method we found that we were not getting the results we desired. This was partially because the method that the Ruge-Kutta-Fehlberg solver was using to adjust the step size had a way of normalizing the error approximation to be within a predictable range and the error approximation in Rosenbrock did not normalize the value. Mainly it comes down to a lack of understanding of the step size adjustment in the Runge-Kutta-Fehlberg solver.

Our investigation had also yielded a different resource for how to implement adaptive step size in Rosenbrock solvers, namely rodas.f. Rodas.f is a freely available Rosenbrock solver written in Fortran, which can be found in the book *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems* (Hairer and Wanner (1996)). This particular solver includes a method for adapting the step size. We tried using the same method as rodas.f to adjust the step size and were able to get the results we desired. Once it included the adaptive step size component, the Rosenbrock solver was able to handle ODEs whose solutions escape to infinity in finite time. Much like the other solvers mentioned here, the Rosenbrock solver will stop short of the singularity as it will reach the minimum step size while trying to solve within the error tolerance. One unfortunate consequence of using this method for handling singularities in the solutions come when plotting orbits that tend to infinity. The message that the solver must stop because minimum step size has been reached is displayed twice, once for the forward direction and once for the backward direction. This is annoying for the user and perhaps in future development this could be changed. An example of this situation is in test 2 of Professor Borrelli's test suite ($x' = 1 - y^2; y' = 1 - x^2$).

After implementing this new feature within the Rosenbrock method we went through the same series of test procedures that we had gone through before. Specifically we tested it with our set of ODEs shown in Appendix A.1, as well as Professor Borrelli's test suite (Appendix A.2). Through testing we realized that we needed to make some small changes. Specifically we had to reconsider what the minimum step size should be. We had originally mimicked the default parameters for the Runge-Kutta-Fehlberg solver, which has 0 as its default minimum step size. This would not work as well for the Rosenbrock solver because of the difference in the way each of the solvers adapts the step size according to the error approximation.

In the Runge-Kutta-Fehlberg solver the error approximation is used to calculate a value Δh_i that is then added or subtracted from the current step size h_i to create a new step size value $h_{i+1} = h_i - \Delta h_i$. In the Rosenbrock solver, the error approximation is used to calculate a ratio r_i between the old step size h_i and the new step size $h_{i+1} = h_i / r_i$. So the old step size value is divided by this factor to find the new step size. Thus repeatedly decreasing the step size in the Runge-Kutta-Fehlberg solver would cause it to go to 0 (within computer precision) much more quickly than in the Rosenbrock solver. This means that using 0 as the minimum step size for the Rosenbrock solver meant it took a long time to realize that the solution could not be approximated within the tolerance. Thus a larger minimum step size was necessary. Through testing different values of the minimum step size on several ODEs in our test cases and Professor Borrelli's test suite. We found that 10^{-12} led to a good balance of quickly determining that the solution escapes to infinity but also accurately solving ODEs whose solutions did not have singularities. Obviously there is no "perfect" value for this because it will vary from ODE to ODE. Also, the user can choose to change this at any time, so our choice of the default is not critical.

5.2.6 Test Results

As mentioned, we had several tests that we ran to validate the quality of our solver and test its robustness. At this point we will present the results of some of these tests to demonstrate the success of the Rosenbrock solver. First, for some of the tests, the analytical solution of the ODE was known so we were able to compare the output of the Rosenbrock solver to the correct solution. Here we will present the results of two tests for which we know the analytical solution. First, the system $x' = y$ and $y' = -x$. We chose the initial conditions $x(0) = 1$ and $y(0) = 0$ so the analytical solution is $x = cos(t)$ and $y = -sin(t)$. Using the solver parameters: absolute Tol

$= 10^{-10}$, relative Tol $= 10^{-7}$, minimum step size $= 10^{-12}$ and maximum step size $= 0.1$, which are the defaults, the error in the solution for x and y from the Rosenbrock solver were on the order of 10^{-7}. When the relative tolerance was increased to 10^{-6} (all other parameters remained the same), only the maximum step size was used and the error was on the order of 10^{-5}. Resetting the relative tolerance to the default and adjusting the absolute tolerance, we found that as the absolute tolerance increased by an order of magnitude, the error increased, but not by a full order of magnitude. There was only a slight increase in the error.

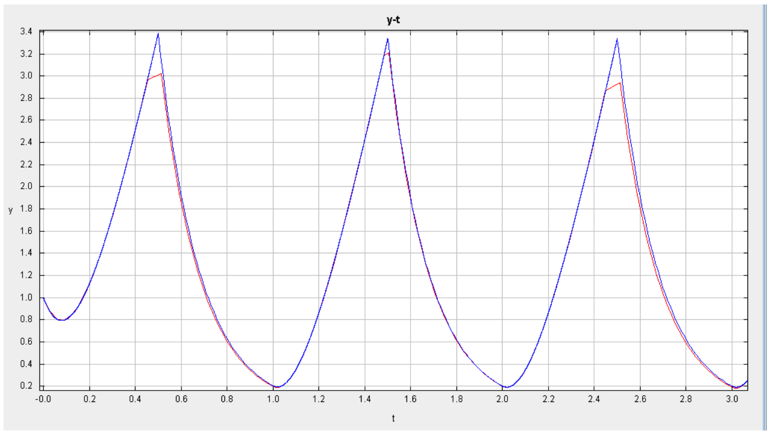


Figure 5.1 Solutions from the Rosenbrock and Runge-Kutta-Felhberg solvers are compared for a non-differentiable ODE

The second test for which we compared the numerical solution to the analytical solution is a stiff system: $x' = 998x + 1998y$ and $y' = -999x - 1999y$. For this system we used the initial conditions $x(0) = 1$ and $y(0) = 0$ so the analytical solution is $x = 2e^{-t} - e^{-1000t}$ and $y = -e^{-t} + e^{-1000t}$. Again, using the default parameters, we found that the error in the solution from the Rosenbrock solver was on the order of 10^{-4} for the early points. The error is larger for lower time because that is the period of rapid change in the solution. Increasing the relative tolerance did not affect the error as drastically for this system as it did for the previous system. However, there was an increase in the error as the relative tolerance increased.

Now we will briefly compare the results from the Rosenbrock solver to results from the Runge-Kutta-Fehlberg solver. In addition to being able to solve stiff differential equations more accurately, we noticed that the Rosenbrock solver is often better than the Runge-Kutta-Fehlberg solver at solving differential equations of the form $y' = f(y,t)$ where f is not differentiable. For example, one of our test cases was the ODE $y' = -5.6 * x + sqw(t, 50, 4)$ (where sqw represents a square wave). Figure 5.1 shows a screen shot of the two solutions overlaid. The red solution is from the Runge-Kutta-Fehlberg solver and the blue solution is from the Rosenbrock solver. Though the solution contains points that are not differentiable as in this case, the way that the Runge-Kutta-Fehlberg solution cuts off the peaks is incorrect.

From our testing we were able to validate the accuracy and functionality of our solver. We were also able to verify that the new solver we implemented improved upon the solvers that already existed in ODEToolkit. We did this by demonstrating cases where the Rosenbrock solver correctly solved ODEs that the old solvers struggled with. Overall, the results are what we desired. Now users will be able to solve stiff differential equations using ODEToolkit.

Chapter 6

User Testing

In the Spring semester, we conducted user testing at Harvey Mudd College to obtain opinions about how we could improve the usability of ODEToolkit and to find bugs that we may have missed. A copy of the user test is included in Appendix D.1. While we knew that we were not going to have enough time to implement the suggestions resulting from the user testing before May, we thought that the results would be extremely useful to any future developers who may work on ODEToolkit.

6.1 Choosing a Method

At first, we had to decide between conducting our user testing in a smaller, more controlled setting like a classroom and a larger, less controlled setting such as a mass-email to the entire student body at Harvey Mudd College. The advantages of conducting the smaller user testing are that we can better understand what the participants are trying to communicate, we can answer any questions they may have, and no outside incentives are necessary. In a small class, the students will be expected to work on the user testing during their class-time. If we were to email the student body however, an incentive would be needed to convince students to take the time to go through our tests. The downside of the smaller user testing however is that we have fewer results and are able to cover less ground testing-wise.

If we were to conduct the user testing by sending the mass email, we would use the chat list known as students-l. This chat list, when emailed, would forward that email to the entire student body of Harvey Mudd College. As an incentive, we considered creating a raffle with an Amazon.com gift certificate as a prize. Since we would have more participants if we chose this method, we would most likely break the test up into several subtests and instruct the students to take each test based on where their last name lies alphabetically. If a student chose, they could take more than one of the three or so available tests to be entered in the raffle multiple times.

6.2 Procedure

The user testing itself consisted of fairly specific directions guiding the users through specific functions of ODEToolkit. The questions/directions varied in their specificity however. For a specific example, the directions could tell the users to click on solve forward a specific number of times after entering a specific ODE. On the other hand, some of the directions were as loose as telling the students to experiment with the graphing and plotting options.

Some of the more specific questions ask things such as "Do you think that there should be more in-depth descriptions of the numerical solvers next to the names of the solvers in the solver options box?" Some of the more general questions ask things like "Was the solve forward button intuitive in its naming? Did what you expect to happen, happen when you clicked on it?"

In order to address any bugs that came up during the user testing, we provided a separate set of instructions. We directed the students to enter any specific error messages they came across, mention if there was an actual error message, and what they did exactly to produce that error message. If the bug did not involve an error message, then we just directed the students to describe in detail what occurred and how it happened and why they disliked what happened.

We ended up deciding to conduct the smaller class testing in Professor Adolph's mathematical biology class. Once that testing was over, we sent out an email to students-l as well as to the email lists of the current core differential equations courses. We created a user testing document to give Professor Adolph's mathematical biology class. We had the second half of the class period to talk to the students, and help them go through our user testing. We were initially told we would have more time however, so our user tests were largely left incomplete. The purpose of this user testing was to obtain suggestions regarding the user interface of ODEToolkit as well as to find any additional bugs that the students may have come across. After a meeting with Professor Adolph, we learned what he had been covering in class and tailored the user testing to better fit his class specifically. We did this by instructing the students taking the test to solve certain Lotka-Volterra Model equations and then answering questions about them.

6.3 Results

One of the main things we learned about ODEToolkit from the classroom testing was that students were tempted to type in multiple parameter definitions on one line (e.g. x=y=5 or x=5,y=7.). We also noticed that some students were having trouble with the zoom function. Some other issues were:

- Students were confused about the naming of ODEs
- Students had mixed opinions about whether the term "solve span" was intuitive.
- One student wanted the program to recognize "e" as the natural logarithm base.
- Students didn't want the clear all button to reset the graph settings (especially the window parameters)
- Students wanted the option to affect things about all graphs.
- Some students wanted to be able to go back and change parameters and resolve from an old curves data.
- One student said that the autoscaling sometimes zoomed out unnecessarily.

Based on these results, we would suggest that future developers of ODETookit do the following things:

- Add an explanation next to the solve span window that can be accessed by clicking on a link.
- Do not allow the name window to appear until after the ODE has been entered.
- Allow the parser to handle multiple parameter definitions on the same line.

- Make it so that the clear all button only reset the curves, not the window frame parameters.
- Make a box that can be checked that will allow changes to a graph to affect all graphs. Graphs that cannot have those changed made to them will simply not be affected.
- Make it so that you can change old parameters and resolve old curves from the data tab.
- Improve the autoscaling.
- We would not suggest allowing e to stand for the natural logarithmic base since exp(1) already serves this function in ODEToolkit. Furthermore, allowing e to stand for the constant will just create confusion when users attempt to use e as a variable.

Professor Bernoff sent out an email to all the Math 45 (Introduction to Differential Equations) students at our request making them aware of ODEToolkit and letting them know that they could use it on their homework assignments.

Chapter 7

Deliverables

Our deliverables for this clinic project are as follows:

- Functional software
- The source code for ODEToolkit
- An updated Java documentation of ODEToolkit
- An updated online tutorial for ODEToolkit
- A Java webstart of ODEToolkit available from CODEE's website
- A library of example ODEs accessible from ODEToolkit
- A final report, poster, and presentation displaying our work and results.

Appendix A

Tests

A.1 Solver Tests

We chose these tests because the analytical solutions are known and for more specific reasons, listed next to each test.

1. $x' = 20 * x$, simple exponential
2. $x' = -25 * x$, simple exponential
3. $x' = y\ y' = -x$, simple coupled system
4. $x' = 32 * x + 66 * y + 2/3 * t + 2/3; y' = -66 * x - 133 * y - 1/3 * t - 1/3$, stiff system
5. $x' = 998 * x + 1998 * y; y' = -999 * x - 1999 * y$, stiff system
6. $x' = -5.6 * x + 28 * sqw(t, 50, 4)$, non-differentiable input [sqw denotes a square wave]
7. $x' = -5.6 * x + 28 * sww(t, 50, 1)$, non-differentiable input [sww denotes sawtooth wave] (analytical solution not known)
8. $x' = 1/t^2$, escapes to infinity in finite time
9. $x' = x^2$, escapes to infinity in finite time

A.2 Test Suite

The following is the test suite provided to us by our liaison Professor Borrelli. The page numbers and figures referenced are from Differential Equations: A modeling perspective by Professors Borrelli and Coleman (Borrelli and Coleman (2004)).

Example	Page	Reason for selection
1. Fig 2.8.1/2	104	Solutions escape to infinity in finite time
2. $x' = 1 - y^2; y' = 1 - x^2$		Same reason
3. Figs 3 and 4	140	Square wave input
4. Figs 1 and 2	143	Same reason
5. Fig 4	150	Polar coordinates
6. Fig 3.1.6	166	3-D
7. Fig 3.4.6	190	Periodic orbit
8. Fig 3.8.2	227	Damped soft spring
9. Fig 6.5.1	387	Planar system
10. Ch. 7 cover and Fig 7.11	449	3-D, Autocalculator
11. Figs 7.2.1/2	455	Direction fields
12. Fig 7.2.8	459	Same reason
13. Fig 7.3.4	469	Planar orbits
14. Ch.8 cover	479	3-D orbits
15. Fig 8.3.3	505	Same reason
16. Fig 1	517	Same reason
17. Fig 9.1.2	523	Polar coordinates
18. Fig 9.1.4	526	Van de Pol
19. $y1' = y2$		
$y2' = 1000(1 - y1^2)y2 - y1$		
$y1(0) = 2, y2(0) = 0$		Stiff IVP for Van der Pol
20. Fig 9.2.1/2	530	Cycle-graph
21. Fig 9.2.3	534	Limit cycle
22. Fig 9.4.6	551	Lorenz Attractor

Appendix B

Solver Reference

These are the descriptions of the solvers and solver parameters used in ODEToolkit, as well as some definitions of terms.

B.0.1 Runge-Kutta-Fehlberg (4/5) Solver

The Runge-Kutta-Fehlberg (4/5) method is an adaptive, explicit, multi-stage method. The errors for the adaptive part of this method are estimated by comparing the results from using the Runge Kutta 4 method and the Runge Kutta 5 method. While the formulas for these methods are complicated, they are used frequently and are very accurate.

B.0.2 Euler Solver

The Euler Method is an explicit, first order numerical method. While it is simple, it is also inaccurate when compared to other numerical methods for solving ODEs. This method works by taking the current estimated point, and adding it to the derivative at that point times the constant step size that has been decided upon. This process is iterated until the desired span of time to be solved over has been reached.

B.0.3 Rosenbrock Solver

The Rosenbrock method is an adaptive, semi-implicit, multi-stage method. The Rosenbrock method can be thought of as a generalization of the Runge-Kutta methods. This is done by adding more parameters into the equations for this method. As a result of this generalization, the Rosenbrock

Solver can solve stiff differential equations. This is something that the Euler Method and the Runge-Kutta methods cannot do well.

B.1 Solver Parameters

B.1.1 Absolute Tolerance

The absolute error refers to the difference between an actual value and its approximation. The absolute tolerance is the maximum acceptable value for the estimated absolute error. If the estimated absolute error exceeds this maximum tolerance, then the next data point is approximated using a reduced step size.

B.1.2 Relative Tolerance

The relative error refers to the difference between an actual value and its approximation, divided by the actual value. In other words, the relative error is equal to the absolute error divided by the actual value of what is being approximated. The relative tolerance is the maximum acceptable value for the estimated relative error. If the estimated relative error exceeds this maximum tolerance, then the next data point is approximated using a reduced step size.

B.1.3 Min/Max Step Size

If either the absolute error or the relative error are above their respective tolerances, then the solver re-solves for the next data point using a decreasing step size. However, the step size can never decrease below the minimum step size parameter. The maximum step size parameter represents the original step size that is taken before it may be decreased due to the absolute and relative errors.

B.1.4 Points/Time and Solve-span

The points/time parameter informs the solver how many data points should be solved for per unit of time. Likewise, the solve-span parameter informs the solver of how many units of time to solve either forwards or backwards.

B.2 Solver-related Definitions

B.2.1 Stiff Differential Equation

In general, stiff differential equations are those such that their derivative depends heavily on the value of the solution. Stiff differential equations are difficult for many numerical solvers to handle unless very small step sizes are used.

B.2.2 Adaptive Numerical Method

An adaptive numerical method is one whose step size depends on the estimation of the error. If the estimated error is above a preset tolerance, then the step size is reduced until the estimated error is below the tolerance threshold.

Appendix C

Developer Tutorial

C.1 Getting Started with Eclipse

Eclipse has been the primary editor used on this project. This chapter descrives how to get an Eclipse project up to speed quickly. To install Eclipse plug-ins, from Help in the menubar click on Software Updates. Then select Add Site and give it the appropriate URL. The URL for the packages we use is listed below. Then select the packages you want and install.

C.1.1 Subclipse

Subclipse is an Eclipse tool which connects an Eclipse workspace to a subversion repository. To install the plug-in, follow the instruction above using the URL: http://subclipse.tigris.org/update_1.6.x/ (or higher version) or alternatively following the instructions at http://subclipse.tigris.org/

Once you have Subclipse installed, you can begin working by selecting File -> Import -> Checkout Projects from SVN. The URL should be for the trunk, or a branch you want.

C.1.2 Castor

To use Castor to generate code for reading and writing a file, first install the Castor plug-in from .jar file at http://sourceforge.net/projects/xdoclipse/files/castor/2.0.4/install-com.pnehrer.castor_2.0.4.jar/download, or go to http://xdoclipse.sourceforge.net/ for instruction and more recent version.

Then to use Castor to generate code, first create a schema file (.xsd) to speficy the XML file format - the file format for ODEToolkit is given in io/ODEFileSchema.xsd. Then to create code, right click at the schema file

(.xsd), select Castor and Generate Java source code. Specify the destination package (io.castor for the current version), then run. There could be warnings because the program does not compile yet, but these errors are unrelated and can be ignored.

Note that if a generated file conflicts with the svn repository, it is possible to disconnect the project from svn first by right-clicking at the project, then select team and disconnect (without deleting the svn reference). Then to reconnect, select team, share project, svn, and validate on connection. To take effect of the newly-generated code, we may need to commit twice, once to update files and the other to add new files.

C.1.3 Copyright Wizard

Copyright Wizard is an Eclipse tool that streamlines the process of adding copyright headers to the source files.

- Install an add-on using the following URL: http://www.wdev91.com/update/
- Once installed, open up the Eclipse preferences (Eclipse > Preferences...), and go to General > Copyright.
- Click 'Add...' to add a new license.
- Specify the copyright label, i.e., 'GNU General Public License v3 (GPL)'.
- Under 'Header text': Paste the contents of the 'file header.txt' file located in the license folder in the repository (../ODE Toolkit/license/file header.txt).
- Under 'License file': Enter "license.txt" for the filename. Paste the contents of the 'license.txt' file located in the license folder in the repository (../ODE Toolkit/license/license.txt).
- Now go to General > Copyright > Headers formats.
- Select Text > Java Source File.
- Leave the default settings, except for 'Blank lines after the header' (enter 1), and hit OK.
- Copyright Wizard is now properly configured. To generate license headers, go to Project > Apply copyright...
- Select the project and hit Next.

- Under 'Copyright type:', select 'GNU General Public License v3 (GPL)'. Under 'File name patterns:' enter "*.java" Check 'Replace existing headers'. Hit Next. Hit Finish.

C.2 Adding a new Solver

To add a new solver, the following changes are needed.

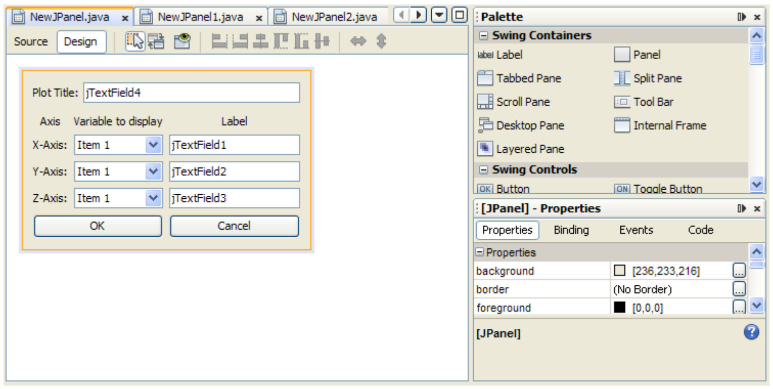
- Add new solver to solver package.
- In SolverSettings.java, add a new SolverSetting (sub)class, and follow the examples of other solvers.
- Modify solve function in SolverInterface.java - follow the examples of other solvers.
- Modify toString in SolverParameters.java - follow the examples of other solvers.
- Add a new panel for the new solver in ui.dialogs.
- Modify SolverChoicePanel.java follow the examples of other solvers.
- Modify SolverOptionsPanel.java - follow the examples of other solvers.

C.3 Use NetBeans to generate UI-related forms

Eclipse does not provide a user-friendly interface for creating Java UI-related form, such as JPanel or JFrame. NetBeans IDE has this feature, so to create a complex UI-related form, we can generate it in NetBeans first, then copy the code for that section to our project in Eclipse. The website for NetBeans is at http://netbeans.org/ - follow the instruction on the website to install the program. In this section we will describe the method for creating a JPanel. However, this can be changed to other Java objects as well.

To create a UI-related form, click File > New File .... Then select Swing GUI Forms > JPanel Form to create a JPanel. Follow the wizard until the file is shown in the editor. The editor consists of Source view and Design view, which can be selected at the toolbar under tab names.

In Design view, we can simply drag Java UI items on the right to the existing JPanel. The interface for this design editor should be intuitive enough to create the desired JPanel form. We can also preview the form by



Plot Title: jTextField4

Axis	Variable to display	Label
X-Axis:	Item 1	jTextField1
Y-Axis:	Item 1	jTextField2
Z-Axis:	Item 1	jTextField3

OK Cancel

Properties

background	[236,233,216]
border	(No Border)
foreground	[0,0,0]

Figure C.1 Form editor in NetBeans IDE.

clicking at the eye icon on the toolbar. The figure shows the construction of JPanel form embedded in Label3DDialog. When obtained the desired form, switch back to the Source view.

In the source view, we need to copy both the initComponents function and the variable declarations (at the bottom of the code) to our class extending JPanel in Eclipse editor. Then we add a call to initComponents() in our constructor of this JPanel. With this, we should be able to run our project in Eclipse with the desired JPanel. We can then add more code and listeners to finish implementing the class.

C.4 Java Web Start - Digitally Signing the Jar

The following steps are necessary for running a jar file using Java Web Start. Perform the following steps in a terminal window:

1. Create a new key in a new keystore:

   ```
   keytool -genkey -keystore <KeyName> -alias <Name>
   ```

   You will then be prompted for a password, first and last name, organizational unit, organization, City or Locality, two-letter country code, and you will be given the option to create a password unique to yourself.

2. (Optional) Create a self-signed certificate as follow (in the directory of your choice):

```
keytool -selfcert -alias <Name> -keystore <KeyName>
```

To list the contents of the keystore, use this command:

```
keytool -list -keystore <KeyName>
```

3. Digitally sign the jar File using jarsigner (the file must be in the same directory as the keystore):

   ```
   jarsigner -keystore <KeyName> file.jar <Name>
   ```

4. Make sure to add the following tag to any jnlp file.

   ```
   <security>
     <all-permissions/>
   </security>
   ```

C.5 Javadoc

Javadoc is a tool for generating API documentation in HTML format from comments in source code. It should be used consistently to document each class as well as all non-trivial functions (excluding, for example, getters and setters). Javadoc syntax can be seen in most classes in the project, and is similar to block comment syntax (/** ... */). Note that these comments must be placed directly before the function or class declaration. To add overview- and package-level documentation, a separate HTML file must be added in the directory of the package, titled package.html (or overview.html). See http://java.sun.com/j2se/javadoc/ for details, and look at existing documentation for examples. For simplicity, please try to consolidate all package documentation into the Javadoc output. Compiling Javadoc is simple with Eclipse: with the root directory of the project selected, select Project -> Generate Javadoc. A dialog will allow Javadoc to be generated for different levels of visibility; for this project, create documentation for members with private visibility. Selecting Next allows the documentation title as well as the overview HTML file to be specified. Eclipse will create the documentation in the root level of the project in a directory called Doc. To view at the documentation, start at index.html.

Appendix D

User Testing

D.1 The User Testing Form

Below is the user testing form that was given to the Mathematical Biology class (Math 118 / Bio 118) during the Spring semester.

Thank you for helping to improve ODEToolkit. Please follow the below instructions and answer the questions as they are brought up. If at any time, you notice a bug, please write it down at the end of the instructions and questions. If you get an error message, please write down the exact error message and exactly what you did to produce that error message. If the bug did not involve an error message, please describe exactly what happened in detail and what you did to produce that result. Please be advised that it is the software, not you, that we are testing. Any difficulties you have with the software are indications that we need to improve the software. Please note that we have attached two screenshots of ODEToolkit along with some labels that may make this user testing easier for you. If at any point you are having trouble, feel free to ask one of us or to consult the online tutorial which can be accessed from the Help menu. Thank you once again for helping to improve ODEToolkit.

Differential Equations Background

What is your background in Differential Equations? (What related classes have you taken?, what concepts have you covered?, etc.):

D.2 Part 1

1. Choose a name for your first system of ODEs. Please note that ODEs must be entered as first order systems in normal form. Additionally, please refrain from using the variable t as a dependent variable as well as using any capitalized variables. Now enter the below competitive Lotka-Volterra equations:

   n1′ = r1 * n1 * (1 - n1/k1 - a*n2/k1)

   n2′ = r2 * n2 * (1 - n2/k2 - b*n1/k2)

   Please use the following parameter values: r1=r2=0.1, k1=k2=100,a=b=0.8. (Note that this will not work if you copy and paste into ODEToolkit. You must retype the system).

2. Click the Enter ODE button.

3. Note that a blank graph has popped up. Now enter the initial conditions below the box where you entered the ODE.

4. Enter the Solve Span, which is the time-range over which orbits will be plotted.

5. Click the Solver Options button in the bottom left hand corner. From here you can choose which solver you want to use. Choose the Runge-Kutta solver.

6. Click on the N2-N1 tab.

7. Click on Graph Options and enable the direction field.

8. Plot an orbit by changing the Mouse Action and clicking on the direction field.

D.3 Questions About Part 1

1. Would you change anything about the order of choosing initial conditions/entering the ODE/choosing the span over which to solve?

2. Any names you enter for ODEs will not be stored until after you have clicked on the Enter ODE button. Once you have done that, you can create and change the names for ODEs. Do you prefer this method over choosing a name for the ODE before you click on the Enter ODE

button? What method do you think would be best for naming the ODEs?

3. Was it clear what solve span meant?

4. Solve for the n1 and n2 nullclines and equilibrium of this system analytically. After you have done this, try and locate them on the plot comparing both population variables with the direction field function turned on. Note that you may have to adjust the tolerances of the solver as well as the minimum and maximum step sizes. You will also have to use the pan and zoom functions to find the correct region of the graph. Once you have done this, please write down what values you used for the tolerances and the step size bounds. You will also want to increase the solve span to about 1000. It should also be noted that you can find equilibrium points by changing the mouse action.

D.4 Part 2

Click on File and select the New ODE Workspace option. Please enter the below ODE:

n′ = r * n * (1 - n/k) with the following parameter values: r=0.1,k=100.

Do not solve forward or backward yet! Click on the graph after having clicked on the Enter ODE button. Note that the initial conditions have changed. Now click solve forward or solve backward. Note that the solve span started from where you clicked, rather than from the original initial conditions you inputted. Now click on the data tab in the output tabs area. You should see your ODE in a folder in the ODEs and curves area of the screen. Click on curve1 to view the data points of that solution in the data window. Note that you cannot see the solution data points of the first ODE you entered from here. You must go to the workspace tabs in the top left corner and choose the first ODE and then repeat the process of clicking on the Data Tab to see the data points of the solution.

D.5 Questions For Part 2

1. Do you like that ODEToolkit switched the initial conditions automatically when you click on the graph or would you prefer that the initial

conditions can only be changed by the user (you) changing them in the Initial Conditions box?

2. Solve for the equilibria of the variable N analytically. After you have done this, locate them again by examining the plot in ODEToolkit. Please describe in detail how you found the equilibria using ODEToolkit.

D.6 Part 3

Please enter a new ODE without selecting the New ODE option. Just delete the old ODE you entered, type another one in, and select the Enter ODE button. Play around with the plot options near the top of the screen. If you right click on the graph window, a handful of additional, useful options will appear (one of which is fixed vs. automatic scaling). Please experiment with these as well.

D.7 Questions For Part 3

1. Recall the Graph Options, Mouse Action options, the AutoFit button, and the Clear button in the plot options area. Did all of the options do what you expected them to do? Were any of the labels unclear? Did anything not work properly? Is there anything that you would change about these buttons?

D.8 General Questions

1. When we have a large number of equations in a system, would you prefer that all graphs be viewable from tabs? While convenient, this would slow down ODEToolkit. Would you prefer to input what specific graphs you would like to see?

2. When you change a graph's options, it only affects that specific graph. Would you prefer that it affects all graphs?

3. If you said yes to question 2, then please answer this question. Some types of graphs have different available options. How would you prefer ODEToolkit cope with the fact that all graph options are linked but different types of graphs can only have certain options?

4. What actions should we allow the user to do to the ODE curves? Currently deletion is allowed on a curve. We also allow changing the color of a curve. Viewing information about solver parameters is useful and it is being implemented, but is it important to allow users to change the parameters and re-solve for a new curve once a new ODE has been entered?
5. What should we allow the user to do to an entire ODE Workspace that would affect all of its curves?
6. If you used the online tutorial, did it answer your questions? What were your questions?

BUGS:
Please describe any bugs you came across below:

ADDITIONAL COMMENTS:
Please provide any further comments if you wish.

D.9 ODE Files

In case you cannot get ODEToolkit to parse the ODE system, please download these files and have ODEToolkit open them. Alternatively, you can use File > Open from Web .. and input the URL.

http://odetoolkit.hmc.edu/downloads/Lotka-VolterraCompetition.ode
http://odetoolkit.hmc.edu/downloads/Logistic.ode

Bibliography

2010. Community of Ordinary Differential Equations Educators. http://www.codee.org/about-us.

2010. Solving stiff ODEs. http://lh3lh3.users.sourceforge.net/solveode.shtml.

Borrelli, Robert L., and Courtney S. Coleman. 2004. *Differential Equations: A Modeling Perspective*. John Wiley and Sons Inc., 2nd ed.

Burdern, Richard L., and J. Douglas Faires. 2011. *Numerical Analysis*. Brooks/Cole Cengage Learning, ninth ed.

CODEE. 1999. *ODE Architect Companion*. John Wiley and Sons, Inc.

Cohen, Scott D., and Alan C. Hindmarsh. 1996. CVODE, A Stiff/Nonstiff ODE Solver in C. *Computers in Physics* 10(2).

Doi, Eric, Steven Ehrlich, Richard Mehlinger, and Andres Perez. 2009. Redesigning ODEToolkit. Tech. rep., Harvey Mudd College Mathematics-Computer Science Clinic. Project sponsored by CODEE.

Hairer, E., and G. Wanner. 1996. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer-Verlag, 2nd ed.

Press, William H., Saul A. Teukolsky, William T Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing*. New York: Cambridge University Press, 3rd ed.

Radhakrishnan, Krishnan, and Alan C. Hindmarsh. 1993. Description and use of LSODE, the Livermore Solver of Ordinary Differential Equations. Tech. Rep. UCRL-ID-113855, Lawrence Livermore National Laboratory.

Shampine, L. F., Jr. R.C. Allen, and S. Pruess. 1997. *Fundamentals of Numerical Computing*. John Wiley and Sons Inc.