5-2015

Partial Differential Equations Resource

Joseph Richard Dorris
jdorris7@vols.utk.edu

Wilson Parker
wparke10@vols.utk.edu

Joey Allen
jallen89@vols.utk.edu

Alex Kotzman
akotzman@vols.utk.edu

Final Report

Team 10 Members:

Joseph Dorris (Team Lead/Reviewer) - jdorris7@vols.utk.edu

Joey Allen (Lead Tester/Designer) - jallen89@vols.utk.edu

Alex Kotzman (Researcher/Lead Presenter) - akotzman@vols.utk.edu

Wilson Parker (Solutions Architect/Lead Report Writer) - wparke10@vols.utk.edu

Customer:

Bruce MacLennan - maclennan@eecs.utk.edu

Executive Summary

Our goal with this project was to build a piece of software capable of visualizing the solution to partial differential equations of numerous variables in 3D space. To that end, we have created the framework and infrastructure for said software. The package comprises an initialization file parser, an expression parser, all of the numerical methods, and a GUI that includes the majority of the various requisite control interfaces, all of which are largely completed. We were able to link the various components of the software and demonstrate a visual output based on a simple test equation. However, the program still requires additional work. We failed to meet some of our customers initial requirements such as the ability to work with up to 50 variables, but we feel that our software framework has the potential capability to support all of these requirements in the future. During development, we also discovered many things that should be added to make this a more useful software, such as the inclusion of more operators. With that in mind, we set up the software framework so that it would be very easy for future users to add more operators (e.g. the trigonometric functions). Also,

some additional functionality in the control GUI is required, namely the ability to change the color of individual variables as well as the ability to slice the 3D plots. All of this being said, the final product PDER-1.0 has tons of functionality and even more potential.

PDE Solver Requirements

1. Description- A piece of software that is capable of 3D partial differential equation solving and graphing.

2. Open Source- Source code is available to the general public.

3. Input- Input is provided by an input text file or graphical user interface (GUI) inputs that specify the variables and equations to graph.

- **3.1.** Variables- Software accepts the following input
  - **3.1.1.** Number- Up to 50 3D variables can be specified. They are defined over the entire simulation space. (They are fields.)
  - **3.1.2.** Type- Both scalar and vector are supported.
  - **3.1.3.** Bounds- Bounds can be specified for both scalar and vector variables.
- **3.2.** Equations
  - **3.2.1.** Meaning- Each variable has a corresponding formula (representing the partial derivative) that describes the value to be added at each grid point at each time step.
  - **3.2.2.** Syntax- The formulas are specified with a programming language-like syntax (e.g., C++, python, MatLab).
  - **3.2.3.** Operators- Supported operators are to include
    - **3.2.3.1.** Four arithmetic operators
    - **3.2.3.2.** The Unit Step

3.2.3.3. Random Number Generator

3.2.3.4. Laplacian (del squared) of scalar field

3.2.3.5. Magnitude and Divergence (del dot) of a vector field

3.2.3.6. Gradient (del) of a scalar field

3.2.3.7. Curl (del cross) of a vector field

3.3. Initial conditions

3.3.1. Constant fields (defined over entire space)- The initial values of all scalar or vector variables can be a constant over the whole space or set to a user-specified function of the (x, y, z) coordinates

3.3.2. Pre-computed fields- The initial values of some or all of the variables can also be loaded from a file, which might have been produced by another program.

3.4. Parameters- It is possible for the formulas to depend on an arbitrary number of real-valued parameters (e.g., rate constants).

3.5. Interaction- Parameters can be changed from the console during the simulation.

4. Processing

4.1. Grid- PDEs are integrated with respect to time over a uniform 3D grid of a size and spacing specified by the user.

4.2. Size- System will work with up to a 100*100*100 mesh.

4.3. Pausing- It is possible to pause the simulation at any time. While paused, the user will still be able to inspect the graph and modify parameters.

4.4. Time step- The time step (delta t) is set at the start of the simulation, and it is possible to change the time step at any point during the simulation.

4.5. Check-pointing- It is possible to dump the entire simulation (all variables and parameters) to file so that the simulation can be restarted from that file.

4.6. Out of Bounds- If a variable goes out of bounds during simulation, an error indicator is displayed.

5. Output

5.1. 3D controls- There will be the following controls for viewing the 3D plot

5.1.1. Zooming- The user can enlarge a subset of pixels.

5.1.2. Panning- The user can move around to different areas of the graph.

5.1.3. Rotating- The user can rotate around a specified axis either the x or y direction.

5.2. Viewing Variables (2.1.1)

5.2.1. Visibility- Have a checkbox for each of the aforementioned variables that selects whether or not it is added to the graph.

5.2.2. Multiple variables- If more than one variable is selected for display then the system displays the "most significant variable" (MSV) in each voxel. The MSV is determined as follows: each variable has a user-specified scale factor. The MSV is whichever variable has the largest scaled value in that voxel.

5.2.3. Scalar Variables- Hues can be assigned to scalar variables. Variable values are mapped to brightness or saturation under user control.

5.2.4. Vector Variables- Vector values are mapped to arrows. It is possible to change these mappings and scale factors during simulation.

5.2.5. Display threshold- If a variable's value in a voxel is below a user-specified display threshold for that variable, then it is transparent.

5.3. Sectioning- The visibility of voxels is controlled by a plane parallel to the display surface. Voxels in front of the plane are invisible and voxels behind it are displayed. This permits displaying the interior of objects. The location of the plane can be changed during the simulation.

5.4. Inspection- During the simulation, the value of a variable at a specific location in space can be inspected by entering its (x, y, z) coordinates.

5.5. Display rate- It is possible to set the real time interval between display updates and/or the number of time steps computed between each display update.

Progress Report

Our project, as described by our customer, was to create a very user friendly and open source program that was able to solve and visualize partial differential equations in 3D space. Solving and visualizing partial differential equations is very common goal for many scientific applications. While it has been implemented in expensive software such as Matlab and Maple, there has yet to be a standard, open-source tool that specializes on user friendliness and flexibility on solving and viewing these equations in 3D space. We had many design choices to make after the requirements document was finalized. We needed to decide on how we would implement the graphical user interface for controlling the 3D graph. A language had to be decided on to develop this program. We needed to decide on how we would visualize the 3D graph. This would involve searching through the available libraries or application programming interfaces (API's) to common open source visualization software that were available and finding the one that could most effectively meet all of our customer's requirements and still be able to be implemented in our limited time frame. The team also

needed to decide on a design structure of our program, so that we could connect all of the work into a well connected system.

Designing the GUI was a straightforward process, as the customer provided a sample plugin for an existing program which provided a 2D partial differential equations visualizer. While our application was to be 3D and standalone, those considerations weren't relevant for the GUI's core design. Thus, the first step in design was to duplicate the essential functionality of the existing visualizer tool. The visualizer proper was non-interactive, a static view controlled only by adjusting the controls for the equation, so the additional dimension would not modify the controls design. Further, the GUI for the existing visualizer was customizable as a function of the overarching program in which the plugin was loaded - an unnecessary feature for a single-use application.

The basic design of the controls was straightforward enough to be duplicated in large part. A variety of meters, buttons, and sliders allowed simple control over the equation. The sliders in particular were surprising but clever devices, allowing an easy overview of a bounded variable. The full array of options were not all necessary, however. For example, the recording options would be better situated away from the main controls in the toolbar, and some of the other functions were only relevant to the program housing the PDE visualization plugin rather than to the visualizer itself.

It's better to have a simple clean GUI than a complex one, better to specialize than generalize. As such, rather than duplicating just the controls that would be needed, we allowed for an arbitrary GUI generated from the input file. It is possible that in the future, when we have a firmer understanding of PDEs, we will need to hardcode some sliders but as it stands it doesn't look to be necessary. For the present, however, there are some hardcoded sliders as testing placeholders until our input parser is operational. The next step in control

design is to superclass our custom sliders with buttons and potentially with a custom meter in order to be able to affect controls as a group.

Keeping with our simple GUI design philosophy, we decided on making use of a launcher window. When first running the program, a window showing only parsing flags and a file selection box is opened. From there, the control window and graph window are dynamically generated based on the selected file. This allows for a clean program flow, returning to the launcher as a hub whenever a new equation needs to be loaded. Further, separating the control panel from the graph as independent windows allows the user to make his or her own decisions in regards to spacing, such as hiding the controls for a demonstration or dedicating a whole monitor to the graph.

Currently, both the control panel and the launcher are basically functional. The control panel, as mentioned, makes use of dummy controls while the input file parser is in development, but the controls work as expected and are designed in such a way that generating them using information from the parser will be simple. The launcher also works, though the toolbar options are currently dummies, as there is nothing for them to do, and there are no flagging options because there are no flags. It does, however, provide its primary functionality and allow the user to browse the file system for input and pass this input to a new control window. The graph works independently, but for the time being, it is not porting properly to the wrapper classes in the main project.

We chose to develop the application in C++ to allow for speed while still retaining an effective suite of libraries. While C++ does not have a default GUI library like some modern languages, the Qt application framework for C++ provides an option just as effective. As such, we were not forced to build our GUI functions from the ground up or create custom graphics. In fact, Qt's use of the native GUI elements provides our application with a native look

regardless of the operating system on which it runs. The sense of familiarity is important for some users, as a program that looks distinctly like it belongs to another operating system could cause concerns over compatibility when the application is in fact completely portable.

For the visualization implementation we researched multiple libraries, and narrowed our choices down to MathGL, OpenGL, VISIT, VTK. We found OpenGL and VTK to be too low level for this project, and decided we would have to implement several methods that were already implemented in other libraries. VISIT is an open source graphics libraries use by several research communities, and is used at ORNL for scientific applications. While it had some useful tools we could have benefitted from, we found it to be too specific and too high level for our project. Finally, we found MathGL to be most suitable for our design.

MathGL is a library for making high-quality scientific graphics. MathGL provides the functionality we need, and is programmed in our desired language C++.The graphics can also be used with QT, FLTK, and OpenGL interface. This will allow us to have a library that performs the animation we need, while not sacrificing an easy and accessible way to develop a interface.

After we decided on which library to use, we next began working on basic and common visualization features. We created multiple sets of data to thoroughly test our methods. We first worked on mapping x, y, and z with scalar values on a static 3D graph. This was one of the tutorials provided on the MathGL website. This process involved everyone installing MathGL on their own systems. We each had different operating systems (MacOS and Linux) so this tested the portability of our software. Being able to run on all these platforms was not specified by our customer but we believe this will add to the quality of our project if we are able to accomplish this task. OSX will be our first priority, though, as that

is the operating system our customer uses. We were able to complete this initial tutorial and create a png of the image below (Figure 1).

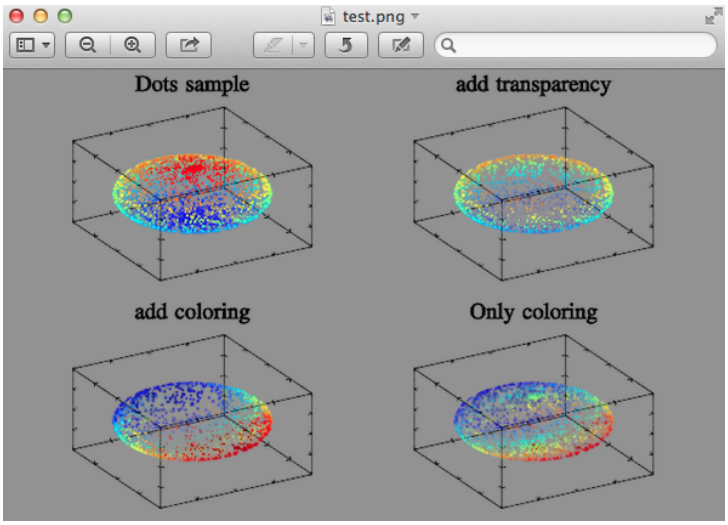


Figure 1

The next task for the visualization was to test the native interactive controls of MathGL to see if they would meet our customers requirements. There are multiple options for interactive windows in MathGL. We chose to try the QT window first because we were using QT for our GUI. This involved re-compiling MathGL with Qt-enable which added some new libraries. We then redid the the initial tutorial but instead of creating a static image, we started a MathGL QT window (Figure 2). This window was then tested for all of the visualization requirements. The zooming, panning, and rotating were all buttons included on the native MathGL window.

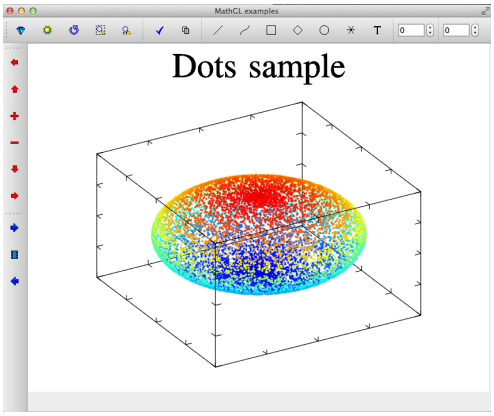


Figure 2

Next, we wanted to make sure that MathGL would be able to create a solid image instead of individual dots that resized with zooming. We found another type of graph supported by MathGL that involved a combination of their 3D surface graph and their 3D contour graph. This looked to be promising so we created the graph below (figure 3) with some help from another tutorial. This also showed how to overlay more than one type of graph, which would be necessary for adding vector variables and representing them as arrows on the same graph. A vector tutorial was then used to add on a vector field to the same graph. This shows an example of what a possible final graph will look like. Another feature that was discovered in this example was the sectioning requirement. A cube would be specified to hide all visualizations that were within the boundaries. With a little tweaking, we found that this feature could be used to slice the graph to look inside the object. We could then add a button that would be able to expand the back wall of the cube backwards or forwards to slice the graph along different planes.

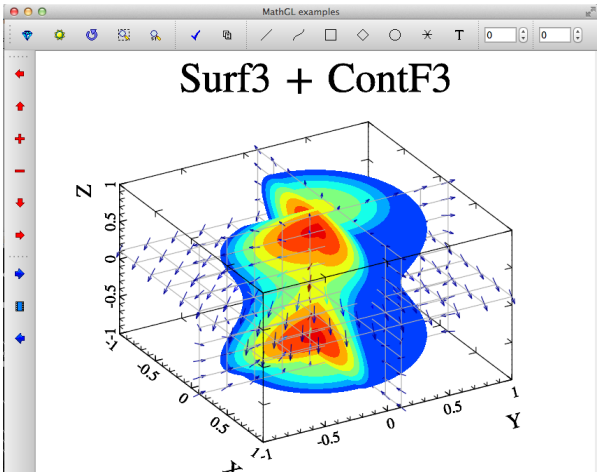


Figure 3

For the animation of the graph, we first attempted to use the QmathGL (MathGL QT) library for developing a running animation in a QTwindow. This proved to be excessively difficult due to the QT requirement of event-handling loops only being allowed to run in the primary thread. Due to this limitation we decided to try out a different library that is integrated into mathGL, FLTK (Fast Light Toolkit). This library allows the user to implement mathGL graphics into a FLTK window. After switching to this library we were able to successfully animate some basic graphs. We used the common method to animate the graph; that is, we ran two threads in parallel. One thread managed keeping up with updating the graph, while another thread managed the calculations. A sample of the of the animation and growth of a Surf3 graph is shown in Figure 4. The frames below show an example from the beginning of the simulation to the final frame of the simulation.

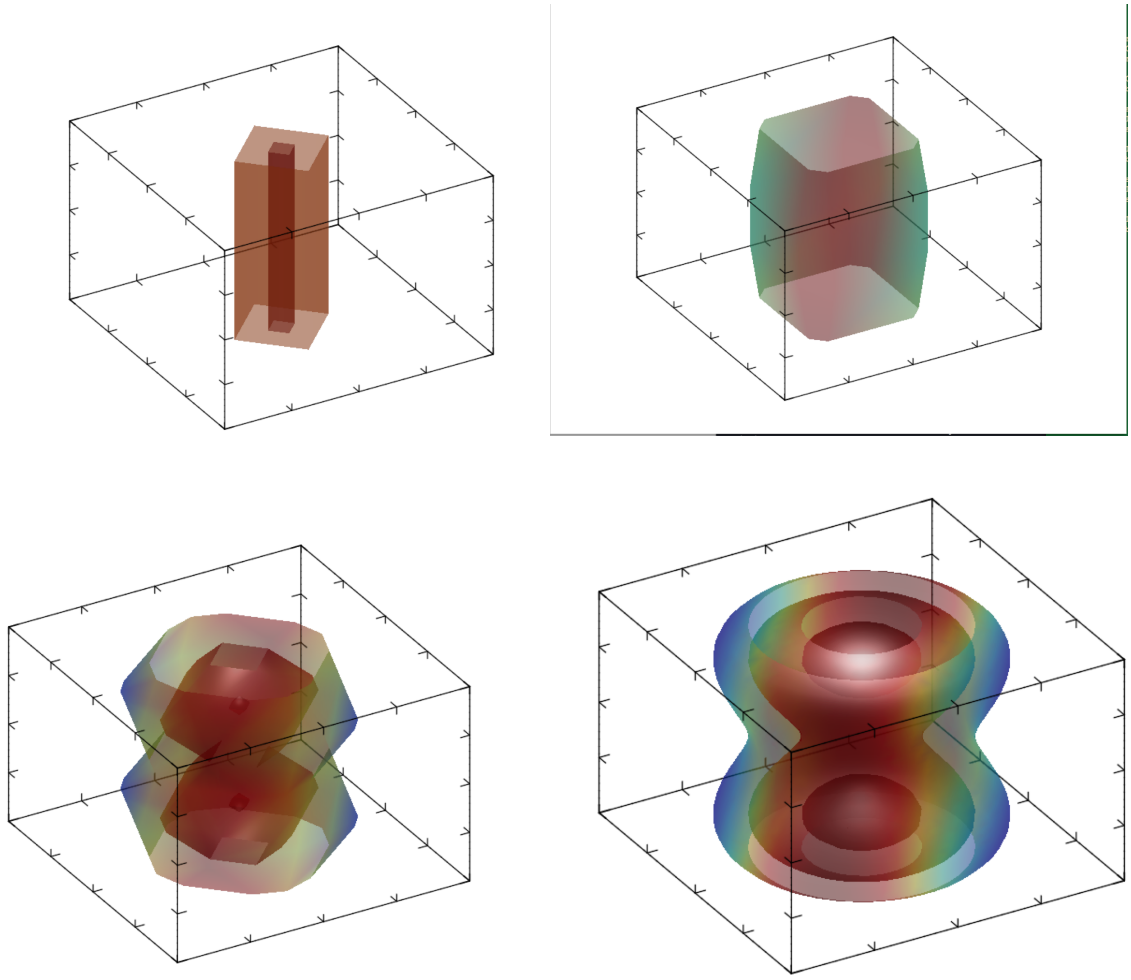


Figure 4

For the overall design of our program, we wanted to adequately break down the problem into manageable pieces so that we could divide the continued work between our team members. We decided on the block diagram shown in Figure 1. Alex Kotzman was going to take the lead for designing the Launcher GUI and Control Panel GUI. Joey Allen will continue the work on Math GL window. Wilson Parker will work on the Text Parser and Preprocessor. Joseph Dorris will then work on the processor.

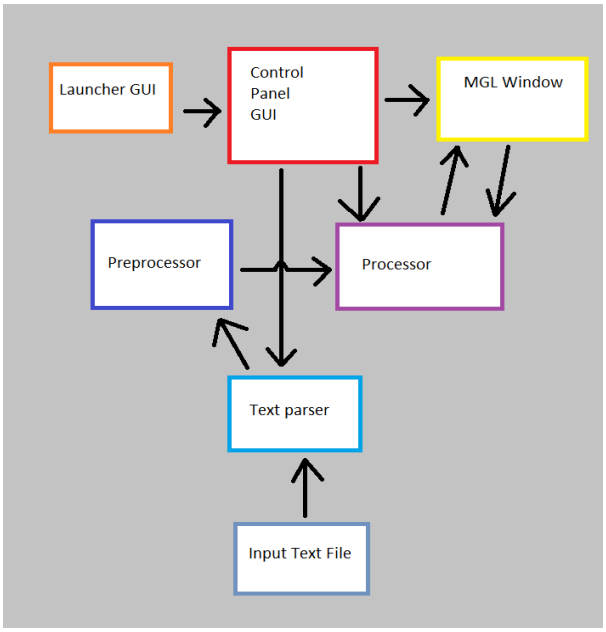


Figure 5

Thus far in the project our main goal has been to set up a basic infrastructure on which to implement the more complicated aspects of the project. Going forward, our main goals are the completion of the numerical methods from which we will extract the various data points to be plotted, the completion of the mathematical operators that are required by our customer to describe the partial differential equations, a file template for equation input, a program capable of parsing and extracting parameters from said file template, and lastly all of the program components must be able to communicate with one another. For the future of animation we need to create a more systematic approach. Right now the animation and drawing functions are intermingled. Ideally we want these separated. This will allow the a user that is not familiar with how create an animation to easily be able to use the animation class. We plan on implementing this method by using several wrapper methods that essentially decrease the amount of features that the MathGL library can implement, but decrease the learning curve and allow other teammates that are not focusing on animation to easily use the methods. To elaborate, an example would be the axis method of the MathGL

library. This method allows the user to adjust the axis in many ways. While this high level of modification is essential and necessary for building a complex library such as MathGL, all the modifications are not required for our project. Therefore the complexity of this method can be reduced by using a wrapper function to statically set some some of the more abstract variables. Doing this will allow a novice user to only adjust some of the minor details to the axis, such as how many ticks, the plotting range, and titles, and will allow the user to ignore more complex variable such as the rotation of the axis and the dimensionality.

The mathematical operators required include a random number generator, a unit step function, the laplacian operator, the magnitude and divergence of a vector field, the gradient of a scalar field, and the curl of a vector field. For these operators we are currently looking into a few libraries that might be capable of implementing them as well as the aforementioned numerical methods. LibMesh is a strong contender at the moment, however it may prove unable to accommodate the very specialized tasks we require for this project. If that ends up being the case, we will resort to implementing the operators using our own C++ code.

For the template file and file reader, we are going to come up with a standard input file template from which the user can insert all potentially pertinent data. The parser will then step through the file extracting all of the data variables, finally sending them to the equation itself. The equation will make use of numerical methods to return a scalar value for each timestep. After the various variables are computed they are compared to generate a color value for each variable in order to represent the variable's significance. The alpha (transparency) of the variable will indicate its significance. After the scalar values and color values are calculated, they are passed to the 3D plotter which will display them for each time step.

In addition to the basic passing of the variables from the input file to the output display, there also needs to be communication between the parameter sliders composing the GUI and the behind-the-scenes variable calculations.

Going forward, half of the team will work on the mathematical methods, which should make up the bulk of the requisite work for this project. The other half will finalize animation, graphics, and GUI. When they finish, they will join in working on the mathematical methods with the rest of the team. As this is going on, the team will take care to ensure that cross-talk between each of the various program components is viable. The final stages of production will consist of linking together all of the program pieces into one solid and efficient package.

Design Document

The partial differential equations resource project has been separated into the following sections. A launcher GUI allows the user to specify an input file. The input file is where the user can list the up to 50 vector and scalar variables and the ranges for these variables. The input file is also where the user can specify the partial differential equations for each variable. It will allow specification of other parameters such space dimensions and time step intervals as well. Translating this input file to the correct output will require a parser and a preprocessor. Once the problem is preprocessed, then a control GUI and MGL window will appear which will allow all of the 3D controls, inspection, sectioning, and display controls specified by our customer. While event handler threads are listening to these two windows, there will be another processor thread that is calculating the solution to the PDE equation for each variable. This processor will contain the numerical methods for each of the required operators and all of the processing requirements. All of these pieces of the project and how they are connected are shown in Figure 1.

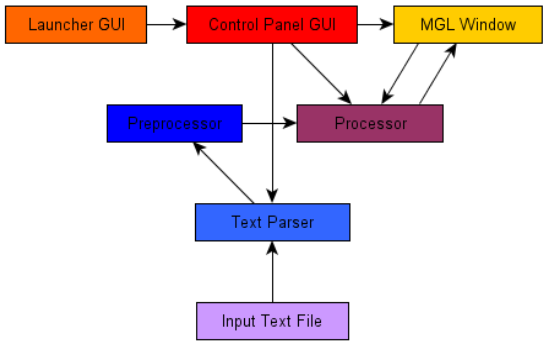


Figure 1

The input file will contain 5 sections: variables, ranges, equations, control parameters, and initial values. A template file will be provided to show all of the functionality provided by this input file and then the user can create their own to be called. Each operator's numerical method will be turned into a function. These functions will be ordered and set up to be called appropriately by the preprocessor to solve all of the equations provided in the input file (Figure 2). This will then set up the calculations for each of the variables so as to be solved for by the processor at each time step. These coordinates and variable values are then passed to MathGL for translation into visualization.

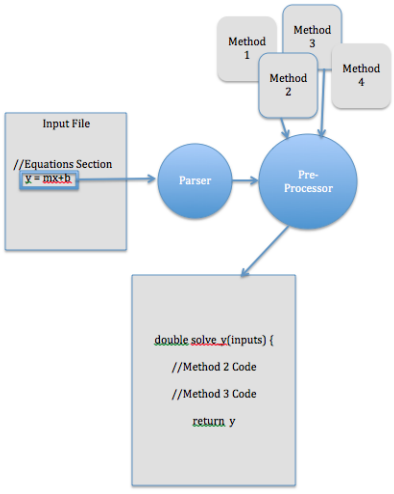


Figure 2

MathGL with FLTK (Fast Light Toolkit) has been selected as the library for implementing the graphics and animation portion of the project. This has been chosen over other options such as Qt (Figure 3) due to its ability to allow for easy communication between the GUI and the underlying mechanisms controlling the drawing and calculations. The best solution currently is to separate the graphics and animation into two different classes. The graphics class will include higher level methods that will allow a developer to control the drawings and calculations. For example the graphics class will include a method for adjusting the brightness of the MGL Window. An example FLTK window is shown below in Figure 4.

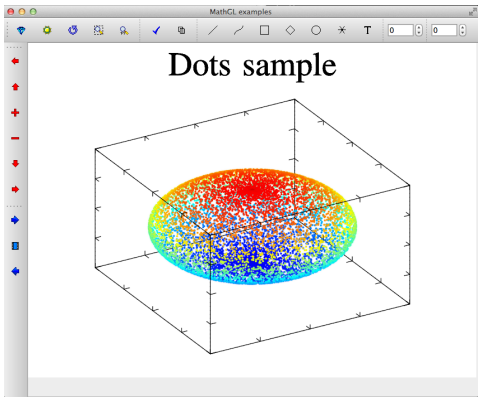


Figure 3

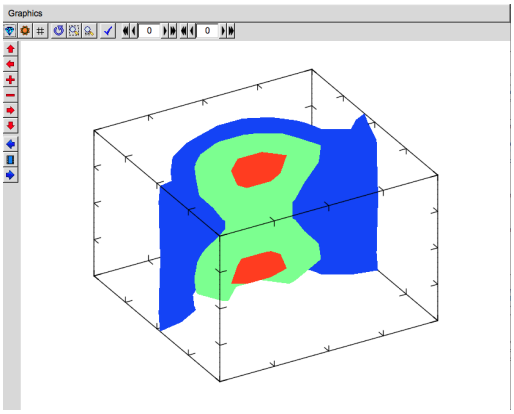


Figure 4

The animation class will focus on controlling the pace and flow of the animation process. It will implement methods that can begin, pause, or stop the animation. It will also allow the user to set the amount of points they wish to draw. The methods provided by the animation class are shown below (Table 1).

Animation Methods	
Public Methods:	Description
beginAnimation	Begins the animation on the MGLwindow
pauseAnimation	Stop Animation on the MGLwindow
getSpeed	Returns the current speed of animation
getPoints	Returns the current amount of points being using for animation.
setSpeed	Adjust the speed of animation.
setPoints	Adjust the amount of points for animation
update	Update the drawing on the MGL Window.
Graphics Methods	
Slice	Allows the user to slice a point in the graph
Brightness	Allows the user to adjust the brightness
Zoom	Allows the user to zoom in on the graph.
Axes	Allows the user to adjust the axis of the graph.

Table 1

We have chosen to separate the work into 4 distinct components for the 4 team members. Alex Kotzman is going to continue working on the 2 GUI's for the launcher and the control panel. Joey Allen will be working on the graphics and animation of the MGL window and communication with the control panel. Wilson Parker will be adding the numerical

methods into the thread that will be doing the calculation (processor). Joe Dorris will be working on parsing the input file, the preprocessing, and the usage of the numerical methods in the processor.

Test and Evaluation of Results

Introduction:

The partial differential equations resource (PDER) has grown into a very large piece of software and will require many tests in order to ensure its functionality and to meet the requirements of our customer Dr. MacLennan. We have decided to test the many components individually in order to perform more tests within a reasonable time frame**.** We went through the list of original requirements and compiled a list of where each of these components should be tested. If any test behaved differently than expected, then the bug had to be corrected. These tests were split up between the team members to be performed separately.

Test Plan (requirement #) -Whether worked successfully

1. Problem Initialization Parser (3)
    a. Test for variable declarations of type vector and scalar (3.1.1)
        i. Up to 50 variables (3.1.1) -N
        ii. Correctly allocates memory for the two types (3.1.2) -Y
    b. Test for constant declarations and the initial value (3.4) -Y
    c. Test for setting the allowable range of these variables and constants (4.1) -Y

    d. Test for equation input (3.2.2)
        i. Test for spaces between characters and no spaces (3.2.2) -Y
        ii. Test for typos (3.2.2) -Y
    e. Test initial value file input
        i. Make sure scalar and vector inputs work (3.3.2) -Y
2. Equation Parser
    a. Test for all operators (3.2.3) -Y
    b. Test for large length of equation (3.2.2) -Y
    c. Operators (3.2.3)
        i. Test boundary conditions (3.2.3) -Y
        ii. Test for zeros (3.2.3) -Y
        iii. Test for all combinations of input variables (3.2.3) -Y
3. Launcher GUI
    a. Test for .pde file extension -Y
4. Command GUI
    a. Try changing variable priority (5.2.2) -Y
    b. Change which vector field to show (5.2.4) -N
    c. Test sliders
        i. Constants (3.5) -N
        ii. Variables (5.2.5) -N
5. Math GL window
    a. Test that all controls work

i. Start/Pause (4.3) -Y

ii. Rotate, Zoom (5.1) -N

iii. Section the graph (5.3) -N

b. Test that all possible signals are received from control panel -N

6. Full Test:

a. Run program -Y

b. Try to open invalid file from file selection window -Y

c. Try to open invalid file with manual text entry -Y

d. Open valid file from file selection window -Y

e. Reload file from manual text entry -Y

f. Run equation for 1 min -Y

g. Try to get results for scalars and vectors from each instance -N

h. Compare results, check for expected changes -Y

i. Try to change scalar/vector order (5.2.2) -N

j. Try to change scalar/vector color (5.2.2) -N

k. Try to change displayed vector (5.2.2) -N

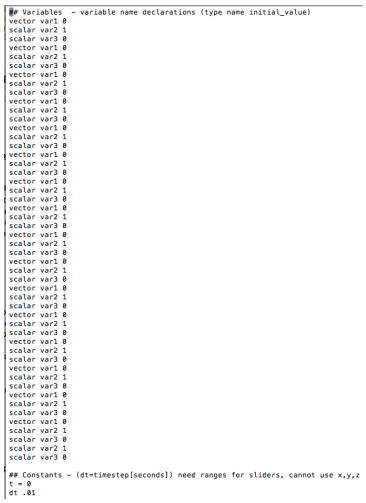
l. Try to hide and reshow scalar (5.2.2) -N

m. Try to unpause animation (4.3) -Y

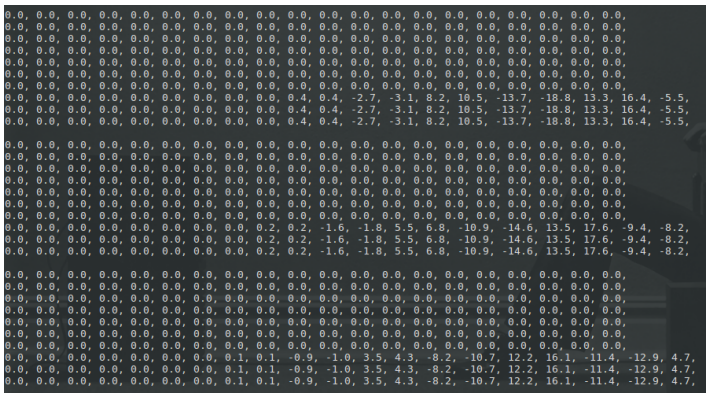
n. Try to rotate graph while animation is playing and while stopped (5.1.3) -Y

o. Try to zoom graph while animation is playing and while stopped (5.1.1) -N

Example tests:



1. Shows input with 50 variables
2. Shows sample command GUI to test



3. Testing the output of specific operators individually (ex. grad())

Evaluation of Results

While our tests did prove the functionality of many parts of our software, we discovered many bugs during the testing process that we were unable to fix in our provided time frame. We were able to meet almost all of our original requirements in regards to the initialization of the problem and solving the equations. The only requirement in this sector we were unable to meet was the requirement to support 50

variables and this could only not be met because of room in the control GUI and supporting that many colors on the Math GL window.

We will continue to correct errors as they arise but by sticking with our original goal of providing ease of use and a wide range of functionality, we introduce a great deal of possibility for user error. Our best strategy for combating this issue with our product is to provide a great deal of documentation and sufficiently commented code to allow for our open source community to enhance the reliability of our software as it is used.

Conclusion

This project was a learning experience for the team. Many of us had yet to collaborate on software of this magnitude. One of our team members, Wilson Parker, was an electrical engineer so had less programming experience but was able to contribute equally by working on the mathematics behind the partial differential equations and numerical methods to solve them. He also led the work on the written deliverables. Alex Kotzman worked on the GUI's and how to connect them to each other. Joey Allen created the visualization window with MathGL for showing the solution to the partial differential equation and the controls for manipulating that window. Joe Dorris was the team lead and wrote the initialization file parser, the expression parser and solver, and the normalizer to create the normalized MathGL data to send to the visualization window. We were then able to combine all of these parts into a viable program. For those interested in using or continuing this project, the repository is on GitHub and is named PDER-1.0.