# Building a MATLAB Graphical User Interface to Solve Ordinary Differential Equations as a Final Project for an Interdisciplinary Elective Course on Numerical Computing

Steve M. Ruggiero*
School of Chemical Engineering
Oklahoma State University
Stillwater, OK

Jianan Zhao*
School of Chemical Engineering
Oklahoma State University
Stillwater, OK

Ashlee N. Ford Versypt†
School of Chemical Engineering
Oklahoma State University
Stillwater, OK
ashleefv@okstate.edu

## ABSTRACT

A final project assignment is described for an interdisciplinary applied numerical computing upper division and graduate elective in which students develop a GUI for defining and solving a system of ordinary differential equations (ODEs) and the associated explicit algebraic equations such as values for parameters. The primary task is to use the MATLAB built-in graphical user interface development environment (GUIDE) [16, 17] to develop a graphical user interface (GUI) that takes a user-specified number of ODEs and explicit algebraic equations as input, solves the system of ODEs using `ode45`, returns the solution vector, and plots the solution vector components vs. the independent variable. The code for the GUI must be verified by showing that it returns the same results and the same figures as a system of ODEs with a known solution. The purpose of the final project assignment is threefold: (1) to practice GUI design and construction in MATLAB, (2) to verify code implementation, and (3) to review content covered throughout the course. The manuscript first introduces the course and the context and motivation for the project. Then the project assignment is detailed. Two student project submissions are described. The verification case study is also provided.

## KEYWORDS

numerical methods, graduate education, computational science elective course

## 1 INTRODUCTION

The corresponding author of this paper teaches a course titled Applied Numerical Computing for Scientists and Engineers that she developed at Oklahoma State University. The course is offered as a chemical engineering upper division and graduate elective designed and advertised to be interdisciplinary. Over the first two offerings, five chemical engineering seniors took the course along with the following numbers of graduate students by degree program: eight in chemical engineering, six in mechanical engineering, two in chemistry, and one each in environmental engineering, mathematics, and plant and soil sciences. The first and second authors of this paper took the course as first year chemical engineering graduate students in the first and second course offerings, respectively.

The course is designed to train science and engineering seniors and graduate students to use practical software tools for computational problem solving and research: Git for version control, LaTeX for mathematical and scientific typesetting, and MATLAB and Python as high level programming languages with libraries of solvers, visualization tools, and capabilities for designing graphical user interfaces (GUIs). Throughout the course, the instructor emphasizes best practices of open-source code development, verification, and documentation [2, 3, 8–15, 18, 20–23, 26, 28] and adopts the Software Carpentry philosophy of providing hands-on training in very practical computational skills for scientists and engineers [27]. Rather than covering the basics of computer programming or details of algorithms for numerical methods, the course focuses on applying numerical computing methodologies, primarily ordinary differential equation solvers and optimization routines for parameter estimation to solve realistic continuum scale problems in science and engineering.

The course consists of ten reading assignments worth 2% each and six computational assignments. The first assignment is worth 5% and introduces the students to using version control in Git and document typesetting in LaTeX, which are required components in all subsequent assignments. The second assignment is worth 5% and gives student practice with programming in MATLAB while developing best practices for scientific computing. The students are required to create a function defining a system of ordinary differential equations (ODEs) and write well-documented code. The third assignment is worth 10% and involves using built-in functions and library routines for numerical methods (specifically ODE solvers) in MATLAB and Python to solve the system of ODEs described in Appendix A. The fourth assignment is worth 15% and covers parameter estimation of dynamic models using MATLAB and Python focusing on a case study from [1]. The fifth assignment is worth 15% and involves creating a relatively straightforward GUI in MATLAB to take user inputs and display simulation results from user-defined functions provided by the instructor for different scientific applications. The sixth and final computational assignment is worth 30% of the course grade and is described in detail in this paper.

*The first two authors contributed equally to this work.
†Corresponding author.


DOI: https://doi.org/10.22369/issn.2153-4136/9/1/4

The final computational assignment is referred to as the "final project" to emphasize its significance in the course grade, its comprehensive nature, and the time involved to complete the assignment satisfactorily. The students are allowed one month to work on this project, and it serves as a take-home final exam. The purpose of this assignment is threefold: (1) to practice GUI design and construction in MATLAB, (2) to verify code implementation, and (3) to review content covered throughout the course. The project builds upon content from all of the previous assignments. Although it does not explicitly involve parameter estimation, the systems of ODEs used for the dynamic models in the fourth assignment from [1] are used in the final project as two of the three verification cases required. For brevity we only provide the most complicated of the verification cases here in Appendix A, which is from the third assignment.

In the chemical reaction engineering course that the corresponding author teaches and many other offerings of the same course around the U.S., the software POLYMATH [25] is used as recommended by the popular textbook [4], which uses the software throughout the book in many examples and homework problems. POLYMATH is a numerical computation package that can solve data regressions and systems of simultaneous ODEs, linear algebraic equations, or nonlinear algebraic equations. These are all of the types of computational science problems encountered in a typical chemical reaction engineering course. In POLYMATH, equations are entered through GUIs in formats that look just like the written form of the equations, so there is not a coding learning curve (Fig. 1 and Fig. 2). After entering equations via GUIs or in lieu of using the GUIs, users can edit code directly in a script file. For details on how to use POLYMATH to enter and solve a system of ODEs, a tutorial document is available on the web [5]. A limitation of the current POLYMATH educational version is that users can enter only 30 simultaneous ODEs and 40 explicit algebraic equations. This is adequate for typical use. However, for the course project in the author's chemical reaction engineering course [6], the limit is often reached. Additionally, POLYMATH cannot combine different types of numerical solvers in the same problem unlike other software programs such as MATLAB and Python. The author originally started programming MATLAB or Python scripts for working around the equation number limit in POLYMATH but found the codes to be challenging for novice users to modify. The final project discussed here grew out of a need to have a POLYMATH-like GUI-based platform built into a more sophisticated software platform when dealing with more than 30 simultaneous ODEs. MATLAB was selected for this final project because of its relative ease in creating GUIs and packaging them as apps, which are very user-friendly for download and installation.

The remainder of this paper provides the required tasks in the final project assignment in Section 2, gives details of two student project submissions in Section 3, and offers conclusions in Section 4. The verification case study is presented in Appendix A.

## 2 PROJECT ASSIGNMENT

In the project assignment, students develop a GUI for defining and solving a system of ODEs and the associated explicit algebraic equations (e.g., temperature dependence of rate constants or values for parameters) for different applications. The primary task is to use the MATLAB built-in graphical user interface development environment (GUIDE) to develop a graphical user interface (GUI) that takes a user-specified number of differential equations and explicit algebraic equations as input, solves the system of ODEs using `ode45`, returns the solution vector, and plots the solution vector components vs. the independent variable. The GUI should work in general for any system of ODEs (specifically initial value problems not boundary value problems). The code for the GUI must be verified by showing that it returns the same results and the same figures as the systems of ODEs defined in Computational Assignments 3 and 4. The more complicated of these verification cases is provided in detail in Appendix A. Students are not penalized if their code does not work for arbitrary ODEs beyond the verification cases.

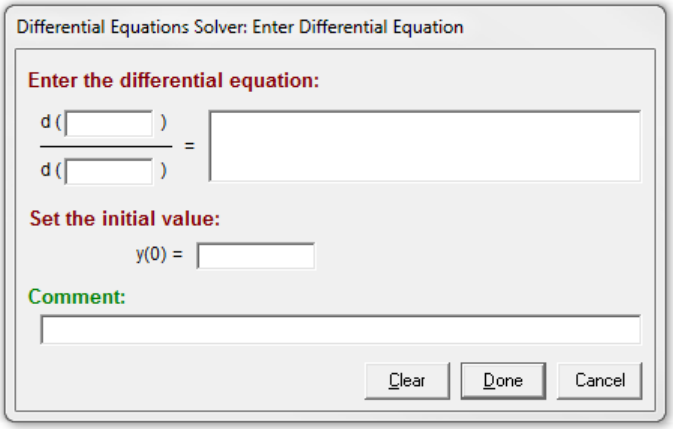


**Figure 1: Graphical user interface in POLYMATH for entering a differential equation.**

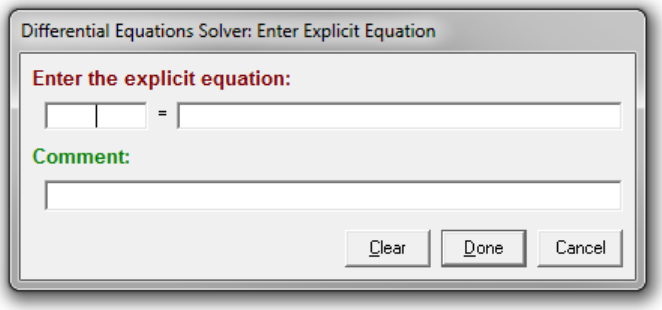


**Figure 2: Graphical user interface in POLYMATH for entering an explicit algebraic equation.**

The GUI can take a variety of layouts, which may involve use of multiple `.m` files, if desired. A descriptive file naming system should be used. Students should design a GUI that is intuitive to use with the following required as buttons:

- define a system of ODEs
  - The numbers of differential equations and algebraic equations should be solicited either directly or indirectly.

    - In the direct method, when this button is clicked, a window should appear asking the user how many equations of both types that they want to define. After that windows should appear recursively for the user to enter the specified numbers of corresponding differential equations and algebraic equations.
    - In the indirect method, when this button is clicked, windows should be opened for defining differential or algebraic equations one at a time (or in bulk for algebraic equations). At the end of each input, the user chooses done or continue to enter another equation of the same type. The number of equations of that type is then incremented.
  - The interfaces for entering the equations should be similar to those used in POLYMATH (Figs. 1 and 2), which can be implemented using one or more windows that are themselves GUIs and/or dialog boxes.
  - The user must be able to enter initial values and limits of integration.
  - The comment section and the buttons labeled *Clear* and *Cancel* shown in the POLYMATH examples (Figs. 1 and 2) are optional.
  - After input is provided by the user, it should be visible to them as a static textbox, editable textbox, or listbox with options to edit the input.
- calculate results
  - This button should calculate results in preparation for either exporting to Excel or plotting.
  - Something should be output to indicate that the results were calculated; command window output is acceptable.
- plot results
  The plot routine should have options for displaying all or a subset of the output vectors (dependent variables vs. the independent variable).
- save plot
  - The save plot button should call `export_fig.m` [19] to create a .png file of a reasonable size for the output plot that currently appears on the screen.
  - The user should be prompted to specify a filename and target directory for the plot.
  - A sample callback function for a button that saves a plot with a prescribed name via `export_fig.m` is shown in Fig. 3.
- export results as Excel file
  The export results button should export all the independent and dependent variable output from the ODE solver to a `.csv` or `.xls` or `.xlsx` file (label the type of output on your GUI) so that the user could view the results in Excel for creating more elaborate plots.

Additional buttons or other GUI objects may be useful. For example, a `.mat` file is recommended for saving selected variables from a subsidiary GUI window for defining the ODEs or the explicit equations. Then this can be loaded in the main GUI file to read the values after the temporary GUI window closes. A cell array is a natural way to save the values generated from each of the GUI windows; however, other ways are acceptable. Another useful button could allow users to edit the axes labels, particularly to include units. The GUI must be packaged as a MATLAB app and submitted electronically via the course website.

```
% Executes on button press in saveplot_button
function saveplot_button_Callback(hObject, ...
    eventdata, handles)
% hObject handle to saveplot_button
% eventdata reserved
% handles structure with handles and user data
if exist('plotName.png', 'file') == 2
    beep
    h = msgbox(...
    'plotName.png already exists. Please rename ...
        or delete the existing plotName.png file ...
        before trying to save again.',...
    'Plot NOT Saved');
else
    ax = handles.plotAxes;
    figure_handle = isolate_axes(ax);
    export_fig plotName.png
    h = msgbox(...
    'The plot was successfully saved as ...
        plotName.png. Be careful to rename it if ...
        you want to save multiple versions of ...
        the plot.',...
    'Plot Saved');
end
```

**Figure 3: `saveplot_button_Callback` function example.**

Students must create a `.tex` file to document testing that their GUI works for the verification test cases. A screenshot of the GUI when all equations have been entered must be included as well as the output plot with all of the dependent variables plotted together vs. the independent variable. A verification case is described further in Appendix A.

String manipulation is not the primary purpose of the final project, but it is necessary to take input from different windows and compose strings into equations. To aid students less familiar with string manipulation, the following tips are provided. If the code can accept the proper input arranged as cell arrays, the function shown in Fig. 4 connects the input to the ODE solver. In test cases, it is clear that the numerator and denominator are not exactly the strings that are needed for `ode45(@(t,y)ODE...)` and that the explicit equations and the right hand sides of the ODEs are not in terms of $y$ and $t$ as desired. It is strongly advised NOT to use `strrep` in MATLAB or other find and replace algorithms. Instead, let MATLAB do that automatically by parsing the strings. Fig. 4 is a working version of a function `ODE` that properly reads cell array inputs stored as variables in `handles` and converts them to the equations that define the system of ODEs. `ODE` is called by `ode45`. The only requirement to the user is that they cannot name a constant parameter $y$ or $t$. This requirement is not necessary for independent or dependent variables.

## 3 STUDENT PROJECT SUBMISSIONS

The top student submissions for the final project from each of the first two course offerings are presented here as examples. We have prepared a private Bitbucket version control repository for

```
function dydt = ODE(t, y, denominators, ...
     numerators, RHSs, explicitEqns)
% Input: t and y are the independent and
%   dependent variable values,
%   denominators, numerators, RHSs, and
%   explicitEqns are cell arrays with
%   the first three terms defining the
%   ODEs of the formă
%   d(numerators) / d(denominators) = RHSs
%   and the fourth term defining the associated
%   explicit equations
% Output: the derivative vector dy/dt for
%   y(1):y(numODEs) where
%   numODEs = length(numerators) = length(RHSs) =
    length(denominators)
% To be called by ode45(@(t,y)...
%   ODE(t,y,denominators,numerators,RHSs,...
%   explicitEqns), tspan, y0)

% independent variable
str0=cell2mat(denominators(1));
eval(strcat(str0,'=t;'));
% dependent variables
for i = 1:length(numerators)
    str1 = cell2mat(numerators(i));
    eval(strcat(str1,'=y(i);'));
end
% explicit equations provided in MATLAB
% acceptable order; can be a
% semicolon separated list
for i = 1:length(explicitEqns)
    str2 = cell2mat(explicitEqns(i));
    eval(str2);
end
% Right-hand sides of ODE definitions
for i = 1:length(RHSs)
    str3 = cell2mat(RHSs(i));
    dydt(i) = eval(str3);
end
% output formatting
dydt = dydt';
```

**Figure 4: Example `ODE` function that properly reads cell array inputs stored as variables in handles and converts them to the equations that define the system of ODEs.**

archiving the code for these submissions along with the instructor documents related to the project assignment. The link is not shared publicly here to prevent future students from simply downloading the solution without doing the project. Interested educators may contact the corresponding author by email to request access to the private repository.

## 3.1 Submission 1

*3.1.1 Overview.* The program of Submission 1 is mainly composed of three parts: collecting user inputs to define the system of ODEs and the explicit equations, solving the system of ODEs coupled to the explicit equations, and saving simulation results by exporting calculated values and figures of plots.

*3.1.2 GUI Description.* The main GUI is composed of ten push buttons, three listboxes, and one axes (plot) object (Fig. 5). The user can provide inputs through the push buttons. List boxes are used to display the ODEs, corresponding initial conditions, and the explicit equations. The axes object is used to display plots of some or all of the dependent variables as functions of the independent variable (Fig. 5 shows only the first dependent variable selected). Multiple GUI windows are used for this program.

All the push buttons, except `Reset` and `Help`, were separated into three groups based on the objectives of the program. Within the panel labeled `Define/Edit Equations/Parameters`, the `Define Equations` push button first allows the user to specify the number of ODEs and gives the user an option to choose whether or not to define any explicit equations after the system of ODEs is defined (Fig. 6). If the radio button `Add explicit equations` is activated (Fig. 6), a GUI window appears that allows user to enter the explicit equations in bulk (Fig. 7) after the system of ODEs has been successfully defined. All the ODEs must have explicit form of $\frac{dy}{dt} = f(t, y)$. Based on the input number of ODEs, a `for` loop is used to repeatedly pop up the window for defining each ODE separately along with its initial value (Fig. 8). When the loop is finished, a new GUI window lets the user define the upper and lower limits of the integration, with both having default values of 0 (Fig. 9). The specified values and equations all appear in the main GUI window either in listboxes or as static text (Fig. 5).

The remaining buttons in the first panel in Fig. 5 are used for editing. The `Edit Selected ODE` push button enables the user to edit specific ODE expressions and the corresponding initial value according to selected line in the `ODE(s) Entered` list box (Fig. 5). By clicking the `Integration Limits` push button, the user can modify the integration limits via the dialog box (Fig. 9). Through the `Edit Explicit Eqs` push button, the user opens a new GUI window (Fig. 7) to define the explicit equations if they have not been defined yet or to edit existing explicit equations.

In the `Calculate/Plot` panel, clicking the push button `Calculate` opens a dialog window to require the user to enter the step size for the numerical integration. The default value is $10^{-4}$. Then `ode45` is called to solve the system of ODEs. When the `Plot Results` button is pushed, the window titled `Select Variables Shown on Plot` appears to allow the user to select single or multiple dependent variables to be shown on the axes of the main GUI (Fig. 10). The user can modify the labels of the independent and dependent axes via the push button `Edit Plot Axes`. The default axes labels are shown in Fig. 11.

The `Save Plot` push button enables the user to specify a file name and target local directory for saving the plot currently shown in the axes area of the main GUI. Similarly, the `Save Data to .csv` push button prompts the user to provide a file name and local directory for saving all the results for the independent and dependent variable output from the ODE solver to a `.csv` file.

The program is capable to some extent of checking for the legality of user inputs. In the window to define a differential equation (Fig. 8), the numerator and denominator positions are checked for the presence of characters and the initial condition blank is checked for a numerical value. The upper limit of integration is required to be larger than the lower limit. When the user decides to edit the notation of the independent variable, the program can only change the left hand side of every ODE; therefore, the user needs to make sure the notation is also modified on the right hand side of each ODE to avoid any errors during the calculation.

*3.1.3 Program Verification.* We entered the system of equations from Appendix A into the program, calculated the results, and

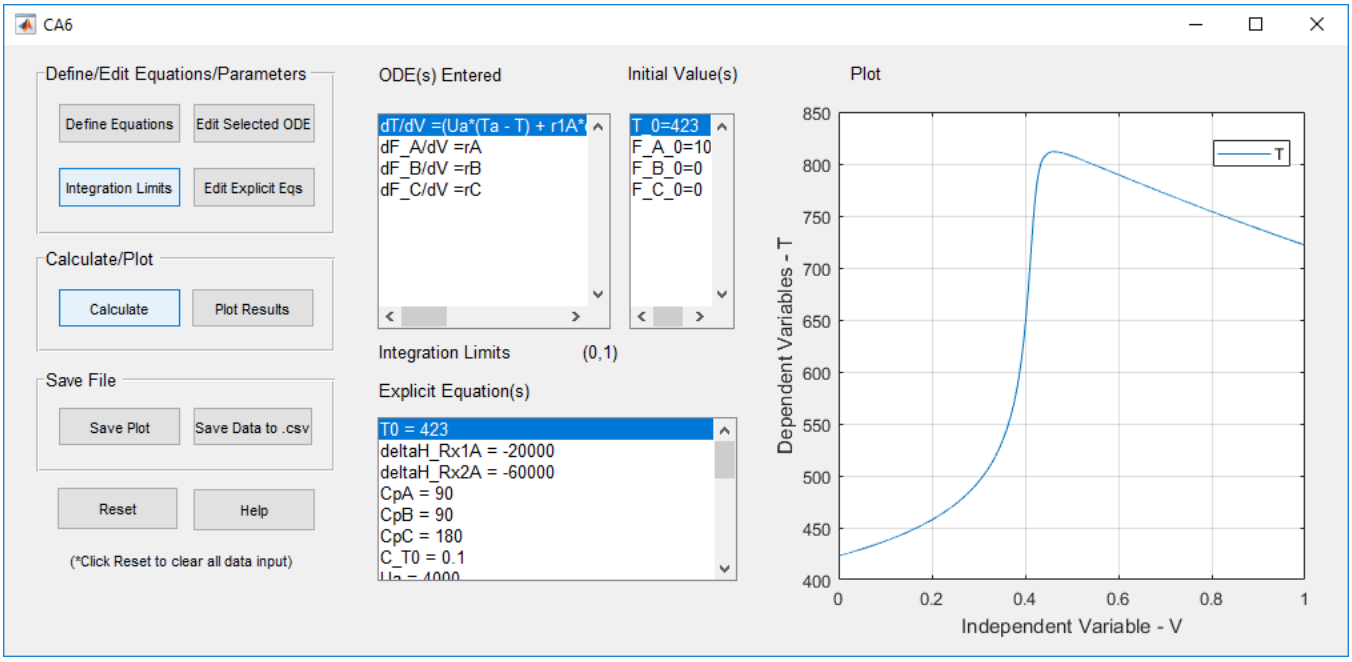


**Figure 5: Submission 1 main GUI screenshot with plot of the dependent variable $T$ for the verification case in Appendix A.**

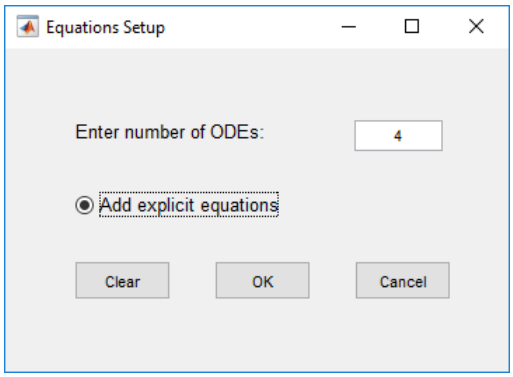


**Figure 6: Submission 1 dialog box for setup of the number of equations.**

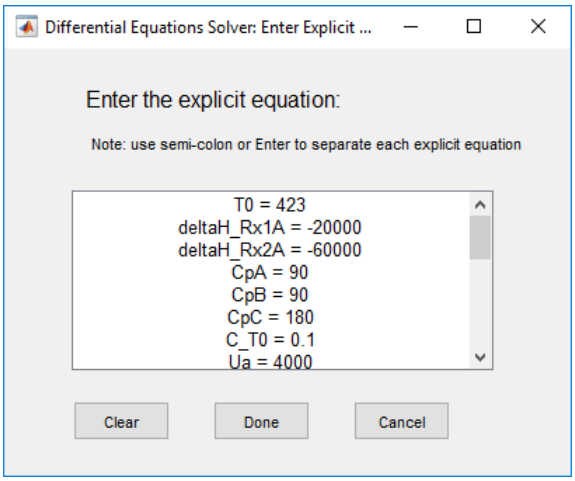


**Figure 7: Submission 1 dialog box to enter the explicit equations.**

plotted two different sets of the dependent variables in Figs. 5 and 12 to see the curves clearly on their different scales.

*3.1.4 Program Shortcomings.* One major defect of this program is that after the number of ODEs is defined at the very beginning, it cannot be changed unless the user redefines the whole system of equations. This program did not utilize the MATLAB GUI `handles` structure for passing arguments between functions. The deficiency related to not being able to edit the number of equations could be compensated by adding another function to manipulate the master `ode_eqs.mat` file, which stores the expressions for all the equations. This could be modified carefully by adding another ODE or deleting one or several existing ODEs. Alternatively, the program could be restructured to utilize the `handles`, in which case an update to the number of ODEs would not be as tricky to implement.

Another defect is that the program is not capable of accepting explicit equations in arbitrary order as in POLYMATH, meaning that a parameter in an explicit equation must be defined before it is used. This requires extra work from the users as all the explicit equations have to be listed in a certain order. This is how MATLAB reads codes, so this is not a serious problem. For possible solutions to this problem, see Submission 2 presented in Section 3.2.

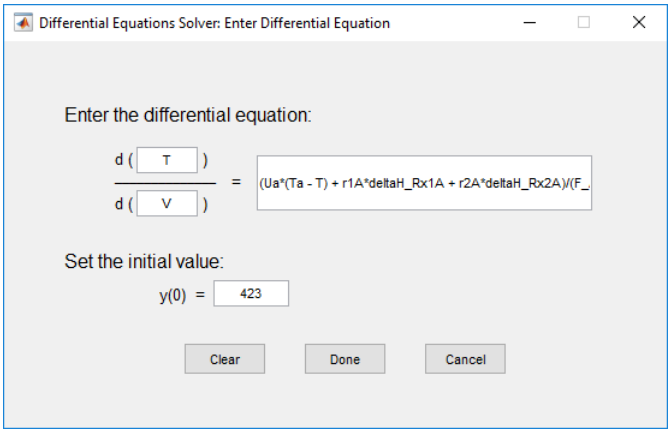


**Figure 8: Submission 1 dialog box to enter a differential equation.**

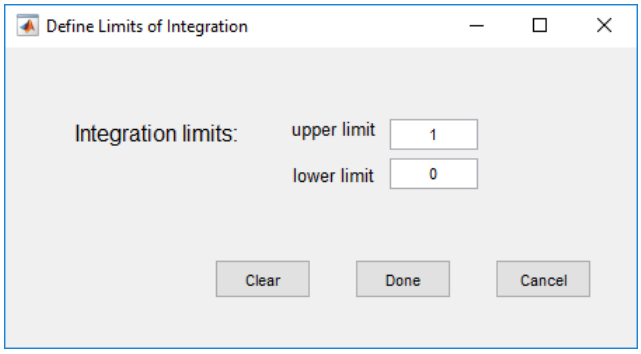


**Figure 9: Submission 1 dialog box to define limits of integration.**

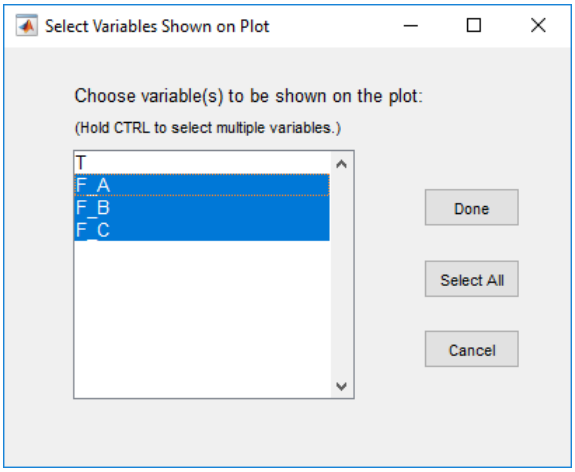


**Figure 10: Submission 1 dialog box to select which variables to display on the plot.**

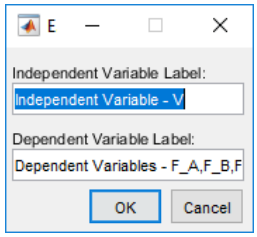


**Figure 11: Submission 1 dialog box to edit axes labels starting from default axes labels.**

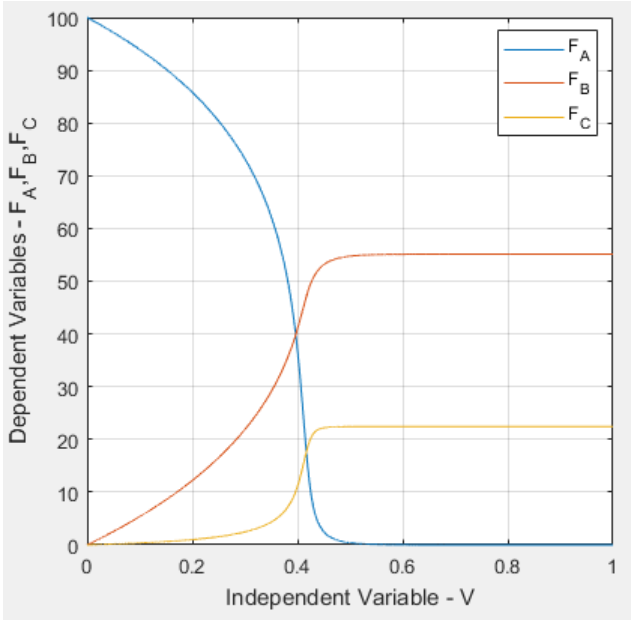


**Figure 12: Submission 1 plots of the dependent variables $F_A$, $F_B$, and $F_C$ for the verification case in Appendix A.**

When the user modifies an ODE, only one ODE from the list box can be selected at one time. Also, items in the `Initial Value(s)` list box can be selected; however, they are not related to any push buttons. In a future version, a new GUI could be added to allow the user to edit initial values based on the selected line in the `Initial Value(s)` list box independent of editing the corresponding ODE.

## 3.2 Submission 2

*3.2.1 Overview.* Submission 2 goes beyond the scope of the project requirements by allowing the user the capability to freely edit equations in the GUI and to enter equations in any order (Fig. 13). Allowing the text to be edited enables the user to wholesale copy text in order to share, save, and enter equations. Additionally, if an error is made in entering the equations into the dialog boxes, the user can quickly fix it by editing the text directly. Furthermore, if the user is comfortable, they can type their equations directly without using the dialog boxes. This is very consistent with the capabilities of POLYMATH.

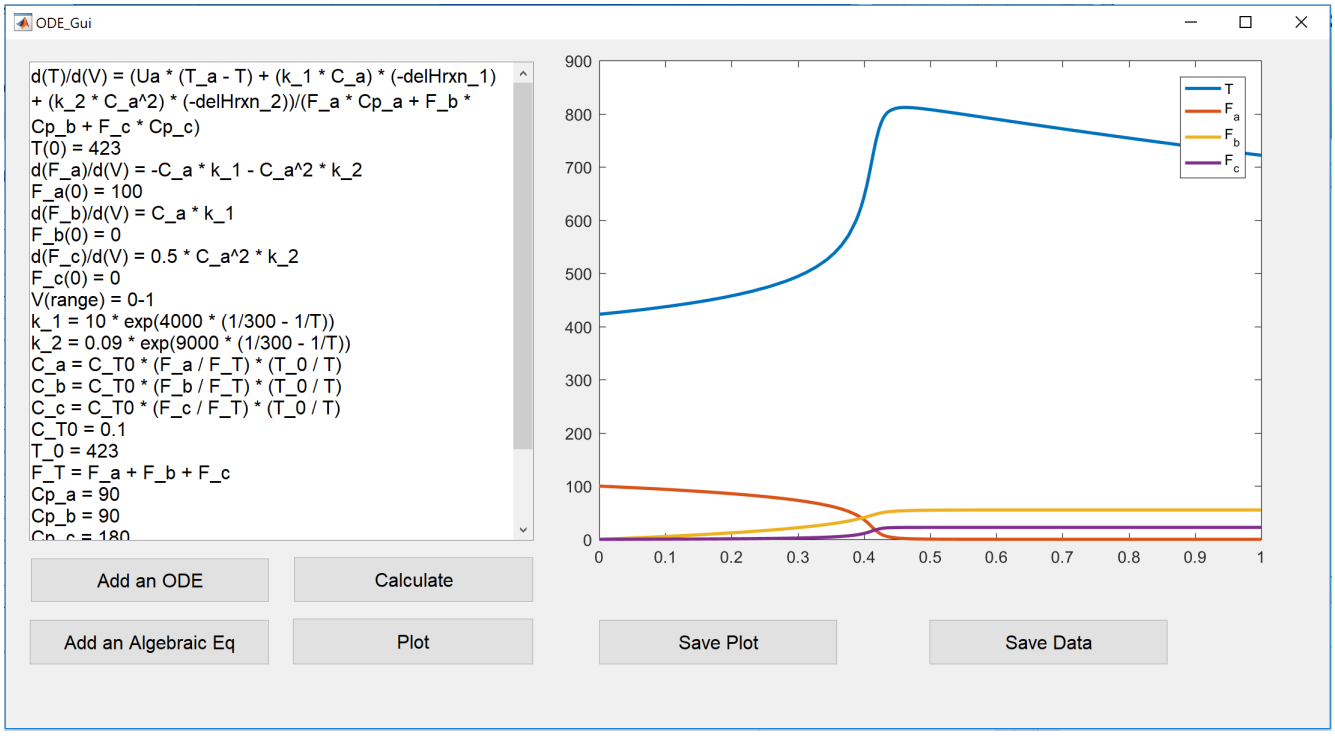


**Figure 13: Submission 2 main GUI screenshot.**

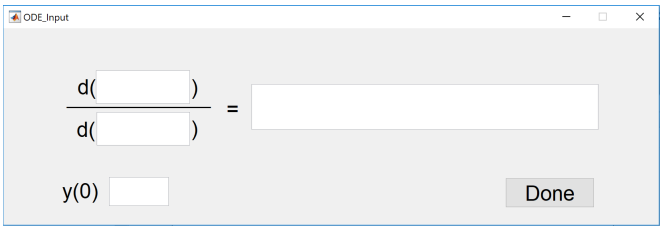


**Figure 14: Submission 2 dialog box for entering ODEs.**

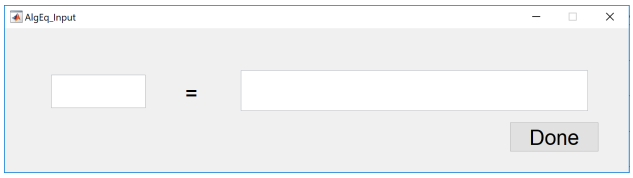


**Figure 15: Submission 2 dialog box for entering algebraic equations.**

*3.2.2 GUI Description.* The main GUI is composed of one editable textbox, six push buttons, and one axes (plot) object (Fig. 13). The user can provide inputs through the push buttons in much the same manner as in Section 3.1 through a GUI for defining a differential equation (Fig. 14) and a GUI for defining a single explicit algebraic equation (Fig. 15). The axes object is used to display plots of some or all of the dependent variables as functions of the independent variable (Fig. 13 shows all of the dependent variables selected). The editable textbox is described in detail in the remainder of this section.

*3.2.3 Equation Parsing.* Properly supporting an editable textbox for equation entry presents some key issues. If the text displayed in the app cannot be edited, then input can be gathered solely from dialog boxes in a very structured manner such as in Section 3.1. Since an editable textbox is a much less structured form of input, interpreting the input becomes a major challenge. The first step in handling the input is to parse the text and convert the text into a structured format. The second step is to reorder the equations so that each equation is only dependent on either no equations or only previously defined equations. This is a requirement because MATLAB requires a variable to be defined before it can be used. The GUI does have dialog boxes that can be used to enter equations (Figs. 14–15); however, these dialog boxes do little more than formatting and inserting the appropriate text into the editable textbox as a template for the user to see how to edit the text.

To parse the text entered into the textbox of the GUI, a custom function `ParseEq.m` accepts a string as an argument and returns a cell array that contains 4 elements. The four elements are the name of the variable the equation solves for, the independent variable associated with the equation if the equation is an ODE, a structure containing the results of parsing the right hand side of the equation, and the type of equation. The right hand side of the equation is returned in two parts: a string of code that can be executed to solve for the dependent variable and a list of variables that need to be defined before the code can be evaluated.

At the heart of parsing the entered text is the use of regular expressions. Regular expressions are useful for finding specific sequences of text. Regular expressions make use of wildcards, whitespace characters, alphanumeric ranges, and more to create a very powerful and flexible syntax for matching text. For example, in the GUI the left hand side of an ODE is in the form of `d(y)/d(t)`, where $y$ and $t$ can be any variables. The regular expression 'd\(\w*\)\/d\(\w*\)' will match the left hand side of any correctly entered ODE.

The first step in parsing the equations is to break the equation into two parts for the left- and right-hand sides. This is done by searching for the '=' character and taking either side as separate strings. Of the two sides, the left-hand side is evaluated first. The left-hand side of each equations has a specific structure–the exact form depends on if the equation is an algebraic equation, ODE, an initial value, or the range of an independent variable. The structured left-hand side of each equation lends itself to being easily parsed through regular expressions.

The main concept used in parsing the right-hand side of an equation is to build a line of code by following order of operations to identify the calculation to be done first, producing the code to evaluate the operation, and then abstracting the evaluated term from the rest of the equation using a token. For example, the string `` `x + y * z' `` becomes `` `x + #1' ``, where `#1` is placeholder for a structure containing the results of parsing `` `y * z' ``. Parsing the right-hand side becomes more complex when handling parentheses. The approach to handling parentheses is to replace the portion of the equation inside of the parenthesis with a token and then recursively calling the`ParseEq.m` function with the replaced text. The result is that for each nested level of parentheses, the function is recursively called until the innermost level of parentheses is reached and evaluated normally.

*3.2.4 Equation Reordering.* The problem of ordering the equations is reduced to a problem similar to Gaussian elimination. A matrix is constructed where each row represents a dependent variable, and each column represents a unique variable needed to define a variable. For each dependent variable, the element in the column of each variable needed to define the dependent variable is set to 1. For example, the equations

$$x = 2 \tag{1}$$

$$y = 3 * x \tag{2}$$

$$z = 4 * y + x \tag{3}$$

yield the matrix (with columns and rows labeled for clarity)

| | x | y |
|---|---|---|
| x | 0 | 0 |
| y | 1 | 0 |
| z | 1 | 1 |

Each dependent variable that is defined by an ODE has an associated initial value previously defined. Therefore, all of the columns associated with variables defined in ODEs are zeroed out right after creating the matrix, representing the fact that the variable is defined. Each dependent variable has exactly one equation in which the variable will show up on the left hand side. When a row corresponding to a variable is zeroed out, the associated equation is put at the bottom of the list of equations, and the column associated with the variable is zeroed out. This process repeats until no changes occur in the matrix after an iteration.

If the matrix is not completely zeroed out, then any rows that are not entirely zeros represent variables that are not properly defined, and an error is returned to the user containing the variables that are improperly defined. If the entire matrix is zeroed out, then the list of equations recorded while zeroing out the matrix is the correct order for the equations so that each equation is only dependent on the previous equation.

## 4 CONCLUSIONS

The project assignment has been used for 21 total students across two offerings of the Applied Numerical Computing elective course at Oklahoma State University. The project has been challenging and thought provoking for the students in the course without being unreasonable and overly time-consuming. Each student has typically visited the instructor's office hours more than once over the one month time period allotted for the project. The instructor has offered assistance with debugging and brainstorming and implementing approaches. The most challenging aspect of the project for most students is connecting the input from a subsidiary GUI window back to the main GUI window. The submissions described here provide two different methods for doing this, and the instructions and tips from the instructor in the assignment and in Fig. 4 suggest another alternative using `.mat` files and cell arrays. Students are encouraged to discuss ideas with their classmates, but the project must be an individual effort. The vast majority of the students have earned an A on the project (all who started early enough to complete all of the required components, including the verification cases). The students have given the project a positive reception as they can clearly see how it connects the prior course content related to numerical solution of systems of ODEs and development of GUIs for scientific applications. The project detailed in this paper can be easily integrated into a variety of computational science and engineering elective or required courses. The content is approachable for both senior undergraduates and graduate students from a variety of disciplines given sufficient background in MATLAB programming and GUI design.

Additional cases studies could be used to adapt the project to other disciplines such as numerical methods, computational physics or chemistry, mathematical biology, and other fields of engineering. These case studies could readily be developed from textbook examples in these fields or published modeling studies such as [1] in petrochemical manufacturing, [7] in computational pharmacology, and [24] in mathematical biology.

## A VERIFICATION CASE

For the verification case study, a system of ODEs is used as defined and solved in an example in a chemical reaction engineering textbook [4]. The equations describe the mass and energy balances for a pair of gas-phase reactions that occur in a plug flow reactor that is operated non-isothermally:

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \qquad -r_{1A} = k_{1A} C_A \tag{4}$$

$$2\,\mathrm{A} \xrightarrow{k_2} \mathrm{C} \qquad -r_{2A} = k_{2A} C_A^2 \tag{5}$$

where $A$, $B$, and $C$ are chemical species, $r_{ij}$ is the reaction rate of the $i$th reaction with respect to the $j$th species, and $k_{ij}$ is the kinetic rate constant for the $i$th reaction with respect to the $j$th species. Pure A is fed at a rate of 100 mol/s, a temperature of 423 K, and a concentration of 0.1 mol/L. The molar flow rates of each species, $F_A$, $F_B$, and $F_C$, and the temperature, $T$, as functions of the reactor volume, $V$, are the quantities of interest. Mole balances on each species $A$, $B$, and $C$ give the ODEs

$$\frac{dF_A}{dV} = r_A \tag{6}$$

$$\frac{dF_B}{dV} = r_B \tag{7}$$

$$\frac{dF_C}{dV} = r_C \tag{8}$$

where $r_i$ is the net reaction rate of species $i$. The initial conditions are $F_A(0) = 100$ mol/s, $F_B(0) = 0$ mol/s, $F_C(0) = 0$ mol/s, and $T(0) = 423$ K.

The corresponding elementary rate laws that describe reactions 1 and 2 from (4) and (5), respectively, are

$$r_{1A} = -k_{1A}C_A \tag{9}$$

$$r_{2A} = -k_{2A}C_A^2 \tag{10}$$

where $C_A$ is the concentration of species $A$. The relative rates are

$$r_{1B} = -r_{1A} = k_{1A}C_A \tag{11}$$

$$r_{2C} = -\frac{1}{2}r_{2A} = \frac{k_{2A}}{2}C_A^2 \tag{12}$$

Equations (9)–(12) are combined to yield the net rates,

$$r_A = r_{1A} + r_{2A} = -k_{1A}C_A - k_{2A}C_A^2 \tag{13}$$

$$r_B = r_{1B} = k_{1A}C_A \tag{14}$$

$$r_C = r_{2C} = \frac{k_{2A}}{2}C_A^2 \tag{15}$$

The gas-phase stoichiometry without pressure drop is used to define the concentration of species $A$ as

$$C_A = C_T(0)\frac{F_A}{F_T}\frac{T(0)}{T} \tag{16}$$

where the total flow rate is defined by

$$F_T = F_A + F_B + F_C \tag{17}$$

The rate constants depend on the temperature through the following Arrhenius functions:

$$k_{1A} = 10\exp\left[\frac{E_1}{R}\left(\frac{1}{T(0)} - \frac{1}{T}\right)\right] \text{ s}^{-1} \tag{18}$$

$$k_{2A} = 0.09\exp\left[\frac{E_2}{R}\left(\frac{1}{T(0)} - \frac{1}{T}\right)\right] \frac{\text{L}}{\text{mol}\cdot\text{s}} \tag{19}$$

The energy balance for the reactor is

$$\frac{dT}{dV} = \frac{Ua(T_a - T) + r_{1A}\Delta H_{Rx1A} + r_{2A}\Delta H_{Rx2A}}{F_A C_{P_A} + F_B C_{P_B} + F_C C_{P_C}} \tag{20}$$

The values for the remaining parameters representing physical constants are listed in Table 1.

To summarize, the system of ODEs for the verification case is given by (6)–(8) and (20) for $\frac{dF_A}{dV}$, $\frac{dF_B}{dV}$, $\frac{dF_C}{dV}$, and $\frac{dT}{dV}$, describing the molar flow rates of species, $A$, $B$, and $C$, in mol/s and temperature, $T$, in K in a non-isothermal plug flow reactor. The reactions are at steady-state but vary spatially along the volume of the reactor, hence $V$ is the independent variable. The explicit equations needed to complete the system of equations are given in (9)–(19) and Table 1.

**Table 1: Values of parameters for the verification case study.**

| Variable | Value | Units |
|---|---|---|
| $E_1/R$ | 4000 | K |
| $E_2/R$ | 9000 | K |
| $C_T(0)$ | 0.1 | mol/L |
| $\Delta H_{Rx1A}$ | -20,000 | J/(mol of A reacted in reaction 1) |
| $\Delta H_{Rx2A}$ | -60,000 | J/(mol of A reacted in reaction 2) |
| $C_{P_A}$ | 90 | J/mol· K |
| $C_{P_B}$ | 90 | J/mol· K |
| $C_{P_C}$ | 180 | J/mol· K |
| $Ua$ | 4000 | J/m$^3$· s· K |
| $T_a$ | 373 | K |

## REFERENCES


[1] J. Ancheyta-Juarez and J. A. Murillo-Hernandez. 2000. A simple method for estimating gasoline, gas, and coke yields in FCC processes. *Energy and Fuels* 14 (2000), 373–379.

[2] J. Carver and G. K. Thiruvathukal. 2013. Software engineering need not be difficult. *Workshop on Sustainable Software for Science: Practice and Experiences, SuperComputing Conference* (2013). http://dx.doi.org/10.6084/m9.figshare.830442.

[3] B. Ekmekci, C. E. McAnany, and C. Mura. 2016. An introduction to programming for bioscientists: A Python-based primer. *PLOS Computational Biology* 12 (2016), e1004867.

[4] H. S. Fogler. 2011. *Essentials of Chemical Reaction Engineering* (1st. ed.). Prentice Hall, Boston, MA.

[5] H. S. Fogler and M. Tikmani. [n. d.]. Polymath tutorial on Ordinary Differential Equation Solver. ([n. d.]). Retrieved January 4, 2018 from http://umich.edu/~elements/5e/tutorials/ODE_Equation_Tutorial.pdf

[6] A. N. Ford Versypt. 2017. Choose Your Own Kinetics Adventure: Student-Designed Case Studies for Chemical Reaction Engineering Course Projects. *Transactions and Techniques in STEM Education* In Press (2017).

[7] A. N. Ford Versypt, G. K. Harrell, and A. N. McPeak. 2017. A pharmacokinetic/pharmacodynamic model of ACE inhibition of the renin-angiotensin system for normal and impaired renal function. *Computers and Chemical Engineering* 104 (2017), 311–322.

[8] T. Hall and J-P Stacey. 2009. Chapter 2: Designing Software. In *Python 3 for Absolute Beginners*. Springer, New York.

[9] D. M. Hamby. 1994. Review of techniques for parameter sensitivity analysis of environmental models. *Environmental Monitoring and Assessment* 32 (1994), 135–154.

[10] A. K. Hartmann. 2015. *Big Practical Guide to Computer Simulations* (2nd ed.). World Scientific, Hackensack, NJ.

[11] F. M. Hemez and J. R. Kamm. 2008. A brief overview of the state-of-the-practice and current challenges in solution verification. In *Computational Methods in Transport: Verification and Validation*, F. Graziani (Ed.). Springer-Verlag, Berlin, 229–250.

[12] D. J. Higham and N. J. Higham. 2005. *MATLAB Guide* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia.

[13] J. M. Kinder and P. Nelson. 2015. *A Student's Guide to Python for Physical Modeling*. Princeton University Press, Princeton, NJ.

[14] J. Kiusalaas. 2005. *Numerical Methods in Engineering with Python*. Cambridge University Press, New York.

[15] C. S. Lent. 2013. *Learning to Program with MATLAB: Building GUI Tools*. Wiley, Hoboken, NJ.

[16] MathWorks. [n. d.]. App Building. ([n. d.]). Retrieved February 15, 2018 from https://www.mathworks.com/help/matlab/gui-development.html

[17] MathWorks. [n. d.]. Create a Simple App Using GUIDE. ([n. d.]). Retrieved February 15, 2018 from https://www.mathworks.com/help/matlab/creating_guis/about-the-simple-guide-gui-example.html

[18] C. B. Moler. 2004. *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics, Philadelphia.

[19] Oliver Woodford. 2013 (accessed November 12, 2016). *export_fig: A MATLAB toolbox for exporting publication quality figures.* https://github.com/altmany/export_fig.
[20] J. M. Osborne, M. O. Bernabeu, M. Bruna, B. Calderhead, J. Cooper, N. Dalchau, S.-J. Dunn, A. G. Fletcher, R. Freeman, D. Groen, B. Knapp, G. J. McInerny, G. R. Mirams, J. Pitt-Francis, B. Sengupta, D. W. Wright, C. A. Yates, D. J. Gavaghan, S. Emmott, and C. Deane. 2014. Ten Simple Rules for Effective Computational Research. *PLOS Computational Biology* 10 (2014), e1003506.
[21] A. Prlic and J. B. Procter. 2012. Ten simple rules for the open development of scientific software. *PLOS Computational Biology* 8 (2012), e1002802.
[22] P. J. Roache. 1998. Verification of codes and calculations. *AIAA Journal* 36, 5 (1998), 696–702.
[23] K. Rother, W. Potrzebowski, T. Puton, M. Rother, E. Wywial, and J. M. Bujnicki. 2011. A toolbox for developing bioinformatics software. *Briefings in Bioinformatics* 13, 2 (2011), 244–257.
[24] S. M. Ruggiero, M. R. Pilvankar, and A. N. Ford Versypt. 2017. Computational modeling of tuberculosis granuloma activation. *Processes* 5 (2017), 79.
[25] M. Shacham, M. B. Cutlip, and M. Elly. [n. d.]. POLYMATH Software. ([n. d.]). Retrieved January 4, 2018 from http://www.polymath-software.com/
[26] V. Stodden and S. Miguez. 2013. Best practices for computational science: software infrastructure and environments for reproducible and extensible research. (2013). http://dx.doi.org/10.2139/ssrn.2322276.
[27] G. Wilson. 2016. Software Carpentry: lessons learned [version 2]. *F1000Research* 3 (2016), 62. https://doi.org/10.12688/f1000research.3-62.v2
[28] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson. 2014. Best Practices for Scientific Computing. *PLOS Biology* 12 (2014), e1001745.