AMATH 581
Final Project:
Numerical Integration of the First Painlevé Equation

Riley Molloy

1 Introduction

I've selected option (2) for my final project, in which I've chosen to apply some various numerical methods from the course and apply them to coursework from another class. As a student of complex analysis this quarter, I've found myself fond of functions in the complex plane, and I thought an experiment in the integration of a differential equation defined over the complex numbers would be a great exercise in applying concepts from both these courses.

1.1 Background and Theory

The Painlevé equations are six second order nonlinear ordinary differential equations which have the property that solutions have singularities which are at most poles and cannot be written in terms of elementary functions. The singularities in the solution are referred to as "movable" singularities, meaning they depend on initial conditions. It turns out that these six equations are the only six second order equations whose solutions have the listed property. In this project, we will only cover numerical methods for integrating the first equation, which is

$$u'(z) = 6u^2 + z. \tag{1}$$

Depending on initial conditions, the solutions to (1) have different "pole-fields," or regions in the plane in which there is an even distribution of poles. In literature as well as throughout this project, only initial conditions for $z = 0$ are presented, which must be real-valued. For most initial conditions, the solution has a pole-field throughout the entire plane. For some, the pole-field is eliminated from two sectors of the plane. And for a specific initial condition case, we can eliminate the pole field from most of the plane. However, in this project, the accuracy of computation is not high enough to work with the special case, so we only look at general categories of initial conditions.

1.2 Numerical Approach

The numerical approaches attempted in this project follow the work of Bengt Fornberg et. al.[1] Their approach had two components: a taylor series-based method for the regions of the solution which are pole fields, and a BVP-based Chebyshev method for the smooth regions of the solution. When I started this project, I set out to attempt both, starting with the pole-field component. This portion ended up taking much longer than usual, so I unable to attempt the smooth-region implementation in the time I had. More on this in the "Results and Further Work." The remainder of this section will cover the approach for the taylor series method used in the pole regions.

The goal of the taylor series method is the accurately integrate through regions with poles, i.e. come very close to singularities which nearby look like $\sim 1/z^N$ for positive integer N. Consider the general second-order ODE problem:

$$y''(z) = f(z, y(z)), \quad y(z_0) = y_0. \tag{2}$$

Of course, we could approximate solutions to this differential equation with Forward Euler in a step-size h

$$y'(z+h) = y(z) + y'(z) + hf(z, y(t)),$$

but we look for higher order accuracy. Note that in the case of a differential equation defined on the complex plane, "step-size" has more complicated definition—each step of h must have a magnitude and direction. Later we will see that some steps are more computationally friendly than others. We choose to pursue a Taylor expansion method because shortly thereafter, we will reconfigure the Taylor expansion to work better for the case of a pole-field. A higher order version of Forward Euler, based on Taylor series, takes the form

$$y(z+h) = c_0 + c_1 h + c_2 h^2 + c_3 h^3 + \cdots + c_n h^n + O(h^{n+1}), \tag{3}$$

where chosen n depends on available resources. The coefficients $c_k = y^{(k)}(z)/k!$ can be found by repeated differentiation of (1), which ends up incredibly convoluted and over-bearing. Instead, we notice from (1) that

$$\frac{d^2}{dh^2} y(z+h) = f(z+h, y(z+h)), \tag{4}$$

which give us a few different opportunities. The first option is to symbolically substitute (3) into (4), giving each further coefficient c_k in terms of the previous coefficients. The second option is a numerical approach. However, to my knowledge, the numerical approach required a significant amount of additional scripting, and I was unable to determine how to implement it in the given amount of time. I will address this further in the "Results and Further Work" section.

After finding these coefficients of the taylor expansion, we rethink the idea of the local approximation by polynomials: since these functions are meromorphic, we can expect to

[1] B. Fornberg, J.A.C. Weideman, A Numerical Methodology for the Painlevé Equations, J. of Computational Physics (2011).

have many problem points in a given area. Therefore, this method transforms the taylor coefficients into a rational function approximation, called the Padé rational form. That is, for $\nu = n/2$,

$$y(z+h) = \frac{a_0 + a_1 h + a_2 h^2 + \cdots + a_\nu h^\nu}{1 + b_1 h + b_2 h^2 + \cdots + b_\nu h^\nu} + O(h^{n+1}), \tag{5}$$

where this is mathematically equivalent to (3), but provides better approximations to the function near poles. We note that for any given c_k, we require that

$$c_{k+1} = \sum_{i=1}^{k} c_i b_{k+1-i},$$

so we can solve the matrix system

$$\begin{pmatrix} c_\nu & c_{\nu-1} & \dots & c_1 \\ c_{\nu+1} & c_\nu & \dots & c_2 \\ \vdots & & \ddots & \vdots \\ c_{2\nu-1} & c_{2\nu-2} & & c_\nu \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_\nu \end{pmatrix} = - \begin{pmatrix} c_{\nu+1} \\ c_{\nu+2} \\ \vdots \\ c_{2\nu} \end{pmatrix} \tag{6}$$

for the vector of bs. For this project, I used MATLAB's partial pivoting using the backslash command to solve the system. However, for higher order polynomials, it may be more worthwhile to use a QR factorization. To find the coefficients in the numerator, we can perform matrix multiplication and addition:

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_\nu \end{pmatrix} = \begin{pmatrix} c_0 & 0 & \dots & 0 \\ c_1 & c_0 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ c_{\nu-1} & c_{\nu-2} & & c_0 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_\nu \end{pmatrix} + \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_\nu \end{pmatrix} \tag{7}$$

It should be mentioned that another method of performing this transformation is by using Wynn's ϵ-method. This is much more difficult to implement natively in MATLAB, however, so we ignore it here.

For the integration path in a section of the plane, Fornberg outlines an algorithm which provides much more computational efficiency than a random outline of paths. This is covered in detail below. The algorithm is based on the efficiency of choosing lower magnitude points for a step than higher ones, since after passing through the region of a pole, more error is collected. The Padé expansion (5) is used to calculate the function's value in five different step directions, which is only slightly more costly than the evaluation of a single direction, in order to choose the lowest magnitude (and therefore most efficient) step. (In fact, this is most of the reasoning behind using a Taylor/Padé expansion.)

2 Implementation

In writing the code for this numerical integration, wrote two different MATLAB files: the first file, path.m, is used as the main file for performing the integration. The second file,

pades.m, is a function file which the main file calls at various instances to calculate Padé coefficients of a given point in the path.

2.1 Taylor Series Method

As covered in the previous section, to integrate the function along any given path, we use (4), substituting in a taylor series like (3) for a chosen number of coefficients and equating on both sides. To do this symbolically, I used Mathematica to define f and u:

```
f[u_, z_] = 6 u^2 + z;

u[h_] = c0 + c1*h + c2*h^2 + c3*h^3 + c4*h^4 + c5*h^5 + c6*h^6 +
  c7*h^7 + c8*h^8 + c9*h^9 + c10*h^10 + c11*h^11 + c12*h^12 +
  c13*h^13 + c14*h^14;

rhs[h_] = f[u[h], z + h];
```

To equate coefficients, I ran the following quick script:

```
Do[
Print[Solve[
  Coefficient[rhs[h], h, i] == Coefficient[u''[h], h, i],
  Coefficient[u[h], h, i + 2]]], {i, 0, 12}];
```

This gave be expressions for each further coefficient in terms of the previous coefficients. As seen in the above scripts, I used a maximum of $n = 14$ for these implementations.

To efficiently call for the Padé coefficients at a point z, as seen in (5), I wrote a function in MATLAB to call at any given point in the main file. The code for this function can be found in the appendix, but I will go over important details here. The function takes in as parameters the following: z, the point in the complex plane at which we will find the coefficients; u, the value the function takes at this point, and u', denoted by "up," which is the value of the derivative of the function at this point z. The code evaluates further coefficients one-by-one, using the formulas directly from Mathematica:

```
c(1) = up;

% further coefficients found using mathematica, based on previous coefs
c(2) = .5*(6*u^2+z);
c(3) = (1/6)*(1 + 12*u*c(1));
c(4) = (1/2)*(c(1)^2 + 2*u*c(2) );
c(5) = (3/5)*(c(1)*c(2) + u*c(3) );
```

and so on, up to c_{14}.

To transform these Taylor coefficients into Padé coefficients, having already vectorized the c_ks in MATLAB, I formed the required matrices as in (6) and (7). The linear system

for the b_k coefficients was solved using the backslash command, and the a_k coefficients were found by straight forward matrix/vector operations.

These Padé coefficients are then flipped (in order to use MATLAB's polyval function in the main file) and returned.

2.2 Step 1: Large Grid

The next step of the integration of (1) is to determine the best path of integration in the complex plane, while attempting to minimize computational inefficiency. As mentioned, I attempted to use the algorithm given by Fornberg. In the process, two different grids are formed, and paths are sketched out between points on these grids. The process begins with the large grid, which in my case was 40×40. It spanned $[-10, 10] \times [10, 10]$ in the complex plane.

The process is as follows: choose a random point on the grid, call this point P_1. Then, starting from the origin and using the Padé expansion from the origin, we take one step in the direction of P_1 (magnitude h, where h is preset), along with four other steps: ± 22.5 and ± 45 degrees to either side. Then, the code determines which of these steps yields the smallest value of $|u|$, and that step is chosen. This is repeated until we have reached with h of P_1 (see the appendix code; this is process is performed within the inner *while* loop of the first *for* loop). When that occurs, a remaining random point $(P_2, P_3, \dots)$ is chosen, and we use the same process to reach it—but this time, instead of starting from the origin, we start from the nearest-by point we've already evaluated. At every single step, the value of u, u', and the Padé coefficients are stored. This whole process is the entire first *for* loop.

For a grid step size of $h = .125$ and a traversal step size of $h = .3$ (meaning for the actual paths, I increased the step size), one plot of the paths taken in the complex plane is shown in Figure 1. Note that this algorithm prevents paths from crossing, and seems to cover the grid efficiently.

2.3 Step 2: Small Grid

Having evaluated all points on the large grid (and more), we now look to use those points and the Padé expansion at each of them to evaluate all points on the small grid. In my case, the small grid was 161×161.

The algorithm works as follows: for each point on the small grid, the nearest-by point on the large grid is found, and the values/coefficients this point on the large grid are used to calculate the desired point on the small grid. Note that there may be computationally more effective ways to perform this process, but this was what seemed to be the most straightforward method in my case. This part of the code is much more straightforward, since all of the information used about the function (i.e. the various Padé) coefficients) had already been found.

After finding the value of u at all points on the fine grid, the results can be plotted and examined.

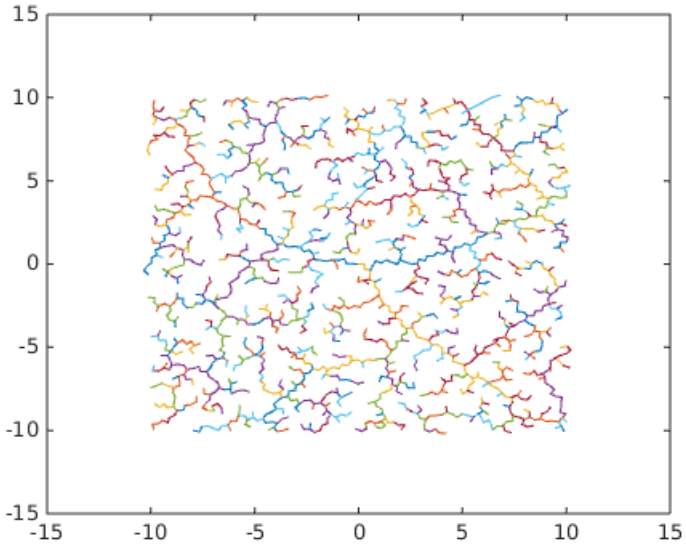


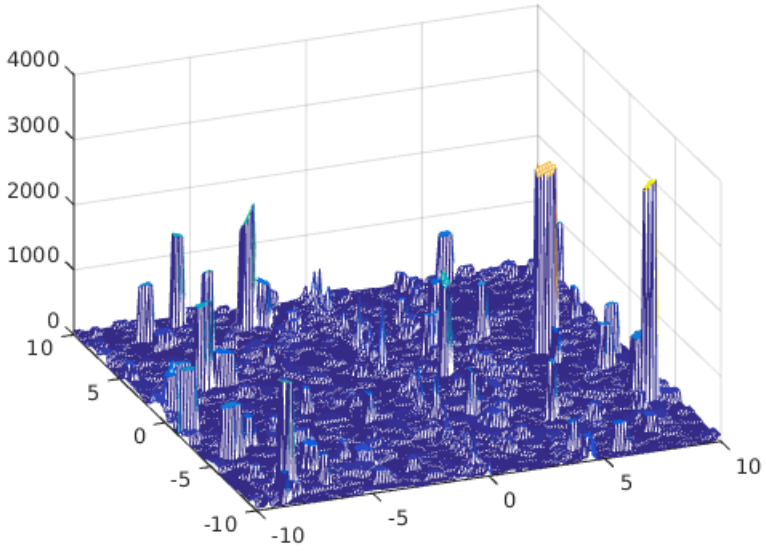
Figure 1: One of the paths of integration taken through the complex z plane to evaluate all of the points on the large grid, as described in the "Large Grid" section.

3 Results and Further Work

Figure 2 shows the results of integration by plotting $|u|$ using the same step sizes as mentioned for Figure 1 (grid $h = .125$, step $h = .3$), for initial conditions $u(0) = -.4$, $u'(0) = .8$. Figure 3 is the same for initial conditions $u(0) = -3$, $u'(0) = 3$. We can notice differences in the values the function takes, where Figure 1 shows some regions with more evidence of a pole field than Figure 2. However, both of these pictures are much rougher than what I was intending. It's possible that this is because my taylor expansion, and hence Padé expansion is relatively low order, when compared to that of Fornberg. I used a Taylor expansion with order $n = 14$, and I believe his was on the order of $n = 30$. Indeed, it's also worth considering that I have an error in my code, or that I've mis-followed the algorithm to give me something that does not make sense.

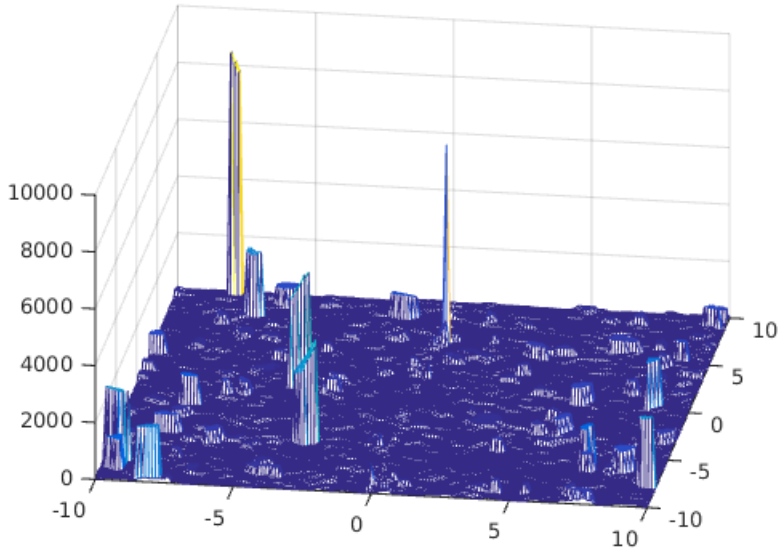
3.1 Automatic Differentiation

As mentioned previously, there are two distinct ways to find the coefficients of the Taylor expansion from (4). The way I found them, for the sake of time, was by finding an analytic expression for each coefficient in terms of the previous ones. The second method would indeed be a more effective method and mostly likely yield better results for sufficiently high order. While Fornberg does not discuss this method in detail, the gained popularity after its



h

Figure 2: A plot of $|u|$ for initial conditions $u(0) = -.4$, $u'(0) = .8$.



h

Figure 3: A plot of $|u|$ for initial conditions $u(0) = -3$, $u'(0) = 3$.

implementation was outlined by Barton, Willers, and Zahar [2]. There are various packages available for Python and C++, though I was unable to find a resource for MATLAB. This method, often referred to as *Automatic Differentiation,* performs a recursive algorithm on a given IVP to generate a Taylor series of any desired order. To further the work I've done in this research, I would look into the implementation of this method in MATLAB, or another language if necessary.

3.2 Analytic Regions

As mentioned previously, this IVP results in different solutions, with different distributions of pole fields, for various initial conditions. In smooth regions, Fornberg solves a BVP, in which the boundaries are the edges of the pole fields or asymptotic regions, using a Chebyshev collocation. With more time, I would implement this part of the solver to get a full visualization of the function.

4 Conclusion

For a complete analysis, there is much work left to be done. Most prominently is the problem of increasing the order of the Taylor/Padé expansion, which would improve the results dramatically. For a more complete view, I would use the skills learned in class for solving BVPs, most notably Chebyshev methods, to solve the smooth parts of the solution. My experience in 581 helped me draw insight to the Taylor series/Padé method, and most notably helped me verify its validity. While I have to conclude my work at this point for submission, I look forward to studying this problem in the future.

[2]D. Barton, I.M. Willers, and R.V.M. Zahar, The automatic solution of systems of ordinary differential equations by the method of Taylor series, The Computer Journal, 1970.

5 Appendix

The two code files have been included in the submission of the assignment.