RobertRash_MATH3316_Project1

September 28, 2017

1 Project 1 - Taylor Series and Floating-Point Error

Robert Rash MATH3316 28 September 2017

1.1 Introduction

1.1.1 Project Structure

```
In [1]: %ls ..
```

```
MATH3316-Project1.xcodeproj/ lib/
Makefile                     res/
bin/                         src/
doc/
```

Excluding the `.xcodeproj` file, which was used for integration with the Xcode IDE, debugging, and profiling purposes, each item in the project directory serves the following purpose:

- `Makefile`: GNU Make project build automation definitions
- `bin/`: compiled binaries. Make will put binaries here by default.
- `doc/`: directory containing all documentation, including this report.
- `res/`: where calculated data is stored after program execution. Files are `.txt` files containing real numbers, space delimited to denote row items, and newline delimited to denote new rows. For this project in particular, the `.txt` files are also organized into separate `res/part{1..3}/` directories.
- `lib/`: reused libraries that are not part of this project specifically. Contains a rewrite of the Matrix library (phrz/matrix).
- `src/`: contains the C++ implementations of the calculations described in this report.

1.1.2 Using this Project

Prerequisites

- A Unix or Unix-like OS (e.g. macOS or Linux)
- A compiler with support for C++14 (LLVM or GNU toolchain)
- Python 3.5
- The latest Jupyter distribution

  - matplotlib

- LaTeX with `pdflatex`
- GNU Make 3.81

Building this project `make all` (default) - will compile binaries, execute them to generate data files, execute Jupyter notebooks with new data files, and convert them to PDFs in `res/reports/`.

`make all_bin` - will compile binaries for part 1, 2, and 3 of this project.

`make all_data` - will compile binaries and execute them to generate data files.

`make clean` - will delete all compiled binaries, generated data, executed notebook copies (but not the original notebooks), and report PDFs.

1.2 Part 1 - Approximation of a Function by Taylor Polynomials

This section covers the computational evaluation of Taylor polynomials, in particular with the function $f(x) = e^x$.

1.2.1 Background

Taylor Series The Taylor series for $f(x) = e^x$ is defined as

$$\sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^n}{n!}$$

This program approximates the above Taylor series and by extension approximates $f(x) = e^x$. As later results will show, these calculations will provide a fairly accurate approximation of the given function.

Absolute Error Absolute error ϵ is defined as

$$\epsilon = |x - \hat{x}|$$

For this project, the absolute error must be calculated for an approximate *function* rather than a simple value x, so the calculation of the absolute error must be done with a function, as well. We can define this function as

$$\epsilon(x) = |f(x) - \hat{f}(x)|$$

1.2.2 Implementation

Horner's Method (polynomial evaluation by nested multiplication) Polynomials, such as the n-degree Taylor series we will be using to approximate e^x, can be efficiently evaluated using Horner's method. These polynomials take the form $p = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$. To perform the calculations along the lines of Horner's method, the coefficients are calculated first and then passed to the `nest` function in a Vector, along with the x value at which that polynomial is being evaluated.

Computation As per the project requirements, the polynomial was evaluated at degrees 4, 8, and 12 ($p_4(x)$, $p_8(x)$, $p_12(x)$) along the linear space [-1.0, 1.0], with increments of +0.01. Each of iteration of these evaluations is stored in `p4.txt`, `p8.txt`, and `p12.txt`, while `f.txt` stores the results of the evaluation of the function $f(x) = e^x$ using the builtin `exp()` function. The computed absolute error for each of the approximations is stored in `err4.txt`, `err8.txt`, `err12.txt`, respectively.

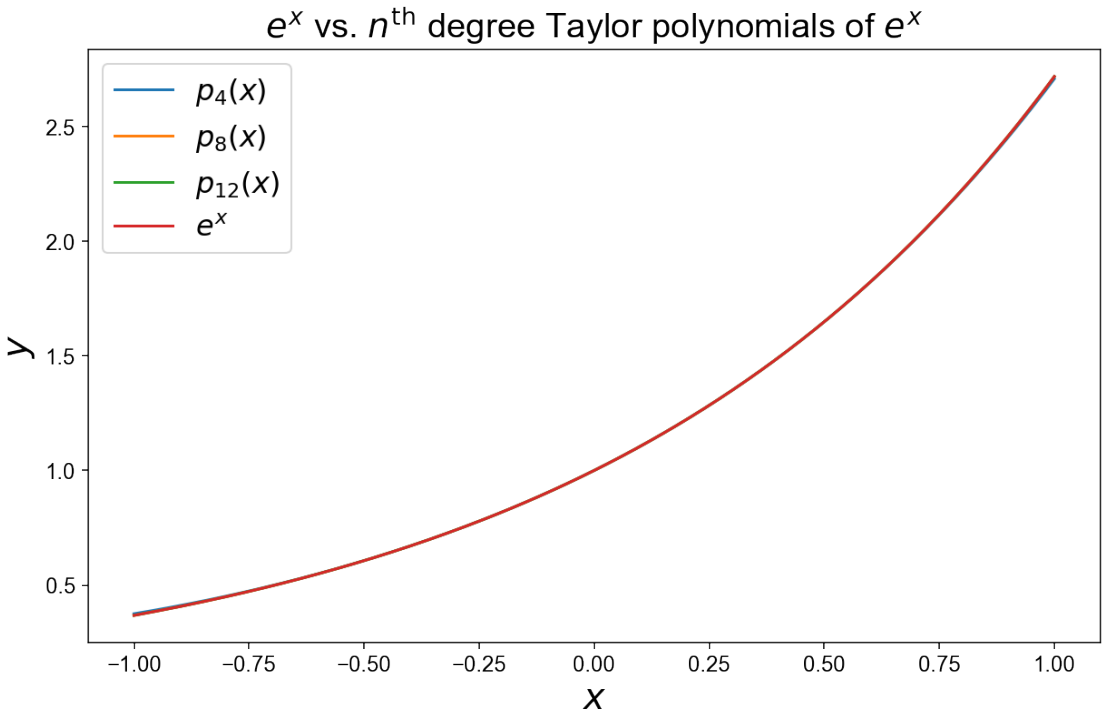
```
In [2]: %pylab inline
        pylab.rcParams['figure.figsize'] = (10,6)
        matplotlib.rcParams.update({'font.size': 16})
        matplotlib.rcParams.update({'axes.labelsize': 20})
        matplotlib.rcParams.update({'xtick.labelsize': 12})
        matplotlib.rcParams.update({'ytick.labelsize': 12})
        matplotlib.rcParams.update({
            'font.family': 'Helvetica, Arial, sans-serif'
        })

        %config InlineBackend.figure_format = 'retina'
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [3]: names = [ 'err4', 'err8', 'err12', 'f', 'p4', 'p8', 'p12', 'z' ]
        data = {name: loadtxt('../res/part1/'+name+'.txt') for name in names}
```

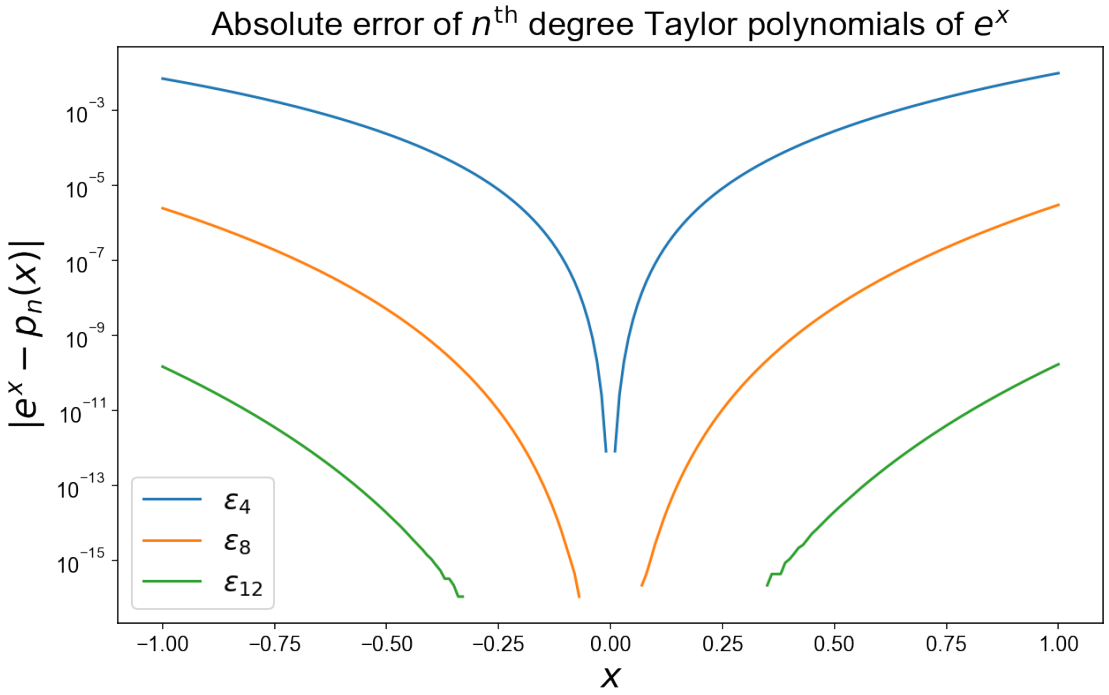
```
In [4]: # Plot e^x, p4(x), p8(x) and p12(x) in the same figure window
        # with different colors. Add a legend, axis labels and title
        # to the plot.

        pylab.plot(data['z'], data['p4'],
                   data['z'], data['p8'],
                   data['z'], data['p12'],
                   data['z'], data['f'])

        pylab.title('$e^x$ vs. $n^{\mathrm{th}}$ degree Taylor polynomials of $e^x$')
        pylab.xlabel('$x$')
        pylab.ylabel('$y$')

        pylab.legend(('$p_{4}(x)$',
                      '$p_{8}(x)$',
                      '$p_{12}(x)$',
                      '$e^x$'))
```

```
Out[4]: <matplotlib.legend.Legend at 0x10bb442b0>
```



In [5]:
```
# Plot |e^x  p4(x)|, |e^x  p8(x)| and |e^x  p12(x)|
# in another figure window using the matplotlib command
# semilogy, again with a legend, axis labels and a title.

pylab.semilogy(data['z'], data['err4'],
               data['z'], data['err8'],
               data['z'], data['err12'])

pylab.title('Absolute error of $n^{\mathrm{th}}$ degree Taylor polynomials of $e^x$')
pylab.xlabel('$x$')
pylab.ylabel('$|e^x - p_n(x)|$')

pylab.legend(('$\epsilon_{4}$',
              '$\epsilon_{8}$',
              '$\epsilon_{12}$'))
```

Out[5]:
```
<matplotlib.legend.Legend at 0x10fc85a58>
```



1.2.3 Analysis

The plotted results of the error calculations above demonstrate that $p_{12}(1)$ is the better approximation of e. As the plot shows, ϵ_{12} is several orders of magnitude smaller than either ϵ_4 or ϵ_8 -- less error correlates to a better approximation.

An error term should be greater than the sum of all approximation terms. That said, this can be done by maximizing all of the possible variables:

$$E_n = \frac{e^{\xi}}{i!} \leq \frac{e^1}{i!}$$

The error term ξ is maximized to 1, as that is the maximum value in the given interval \$ z = [-1.0,1.0]\$. Basically, $\xi \geq \max(z)$.

Then, the degree iterator i is maximized by setting it to $n+1$. The Taylor polynomials only go up to n (where $n \in \{4, 8, 12\}$). Therefore, $n+1 > n$, and thus maximizing i.

$$E_n = \frac{e^1}{i!} \leq \frac{e^1}{(n+1)!}$$

In [6]:
```
import math

E_4 = math.exp(1) / math.factorial(4 + 1)
E_8 = math.exp(1) / math.factorial(8 + 1)
E_12 = math.exp(1) / math.factorial(12 + 1)

print('E_4 = ', E_4)
```

```
        print('E_8 = ', E_8)
        print('E_12 = ', E_12)
```

```
E_4 =  0.02265234857049204
E_8 =  7.490856008760596e-06
E_12 =  4.365300704405942e-10
```

As the results above show, the upper bound calculations are consistent with the error calculations seen in the plot:

$$E_4 >> E_8 >> E_12$$

1.2.4 Conclusion

The results of this program are a good demonstration of the behavior of Taylor polynomials in terms of approximating functions and of the behavior of floating point underflow. As the graphs illustrate, the accuracy of the polynomial approximations increases alongside an increasing degree.

1.3 Part 2 - Errors in a Forward Finite Difference Approximation

This section covers forward difference estimates, which can be used to approximate derivatives.

1.3.1 Background

Forward Finite Difference Estimate The forward finite difference estimate for $f'(a)$ is defined as

$$\delta^+ f(a) = \frac{f(a+h) - f(a)}{h}$$

Where h is the increment.

1.3.2 Implementation

Input Functions There are three functions needed to perform the forward difference estimate and error calculations: $f(x)$, $f'(x)$, and $f''(x)$. These are:

$$f(x) = ln(x) f'(x) = \frac{1}{x} f''(x) = -\frac{1}{x^2}$$

These are implemented as functions `f`, `f1`, and `f2`.

Forward Difference Estimate The forward difference estimate is implemented as a function with two inputs. It implements the following arithmetic:

$$\delta^+ f(a) = \frac{f(a+h) - f(a)}{h}$$

This is implemented as the function `forward_difference_estimate`.

Error Calculation The relative error function is implemented as described in the project guidelines:

$$r = \left| \frac{f'(a) - \delta^+ f(a)}{f'(a)} \right|$$

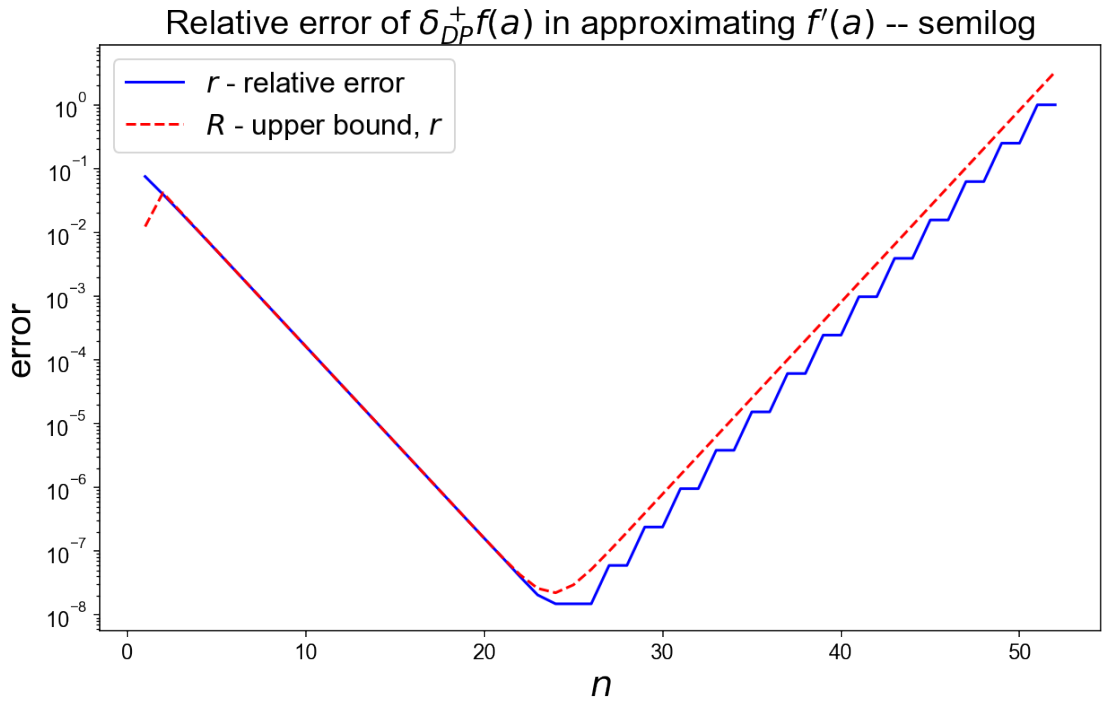
This is implemented as the function `relative_error`. The output is saved in `r_e.txt`.

The upper bound of the relative error is likewise implemented as described in the project guidelines:

$$R = c_1 h + c_2 \frac{1}{h} = \left| \frac{f''(a)}{2f'(a)} \right| h + \left| \frac{f(a)\epsilon_{DP}}{f'(a)} \right| \frac{1}{h}$$

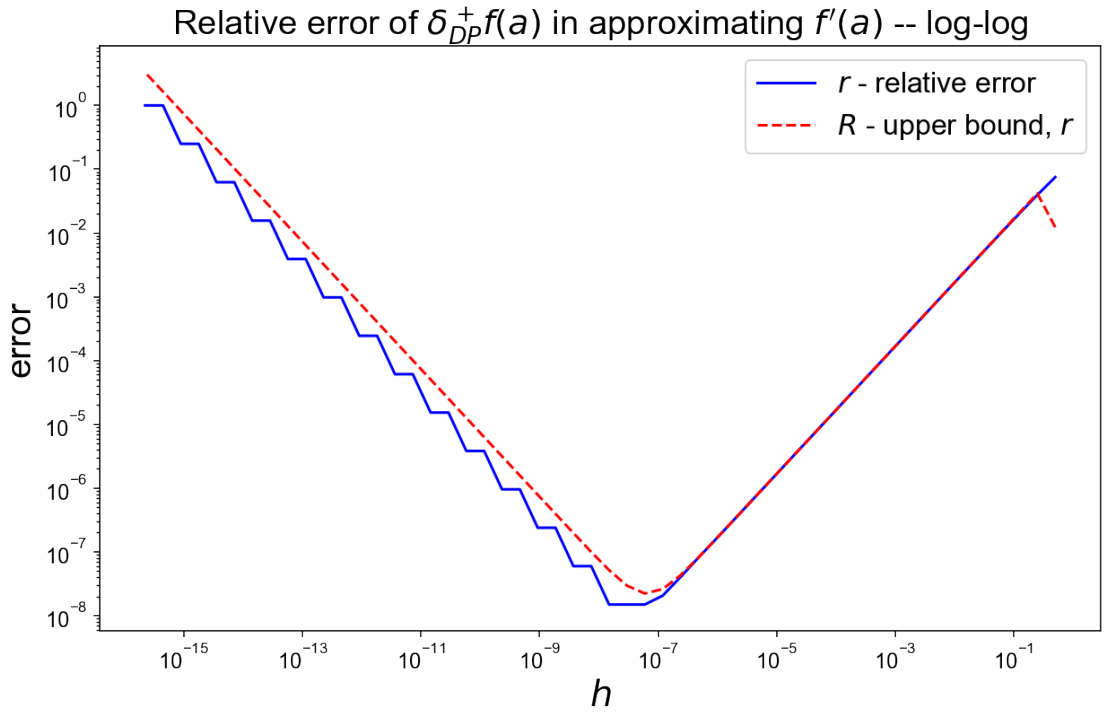
Accordingly, this is implemented as the function `relative_error_upper_bound`. The output is saved in `r_u.txt`.

```
In [7]: %pylab inline
        pylab.rcParams['figure.figsize'] = (10,6)
        matplotlib.rcParams.update({'font.size': 16})
        matplotlib.rcParams.update({'axes.labelsize': 20})
        matplotlib.rcParams.update({'xtick.labelsize': 12})
        matplotlib.rcParams.update({'ytick.labelsize': 12})
        matplotlib.rcParams.update({
            'font.family': 'Helvetica, Arial, sans-serif'
        })

        %config InlineBackend.figure_format = 'retina'

Populating the interactive namespace from numpy and matplotlib


In [8]: names = [ 'n', 'h', 'r_e', 'r_u', ]
        data = {name: loadtxt('../res/part2/'+name+'.txt') for name in names}

In [9]: # Create a semilogy plot that overlays r verus n
        # with a solid blue line, and R versus n with a
        # red dashed line.

        pylab.semilogy(data['n'], data['r_e'], '-b')
        pylab.semilogy(data['n'], data['r_u'], '--r')

        pylab.title('Relative error of $\delta^+_{DP}f(a)$ in approximating $f\'(a)$ -- semilog'
        pylab.xlabel('$n$')
        pylab.ylabel('error')

        pylab.legend(('$r$ - relative error',
                      '$R$ - upper bound, $r$'))

Out[9]: <matplotlib.legend.Legend at 0x1101f01d0>
```



In [10]:
```
# Create a different loglog plot that overlays
# r verus h with a solid blue line, and R
# versus h with a red dashed line. For both
# plots, add a legend, axis labels and title.

pylab.loglog(data['h'], data['r_e'], '-b')
pylab.loglog(data['h'], data['r_u'], '--r')

pylab.title('Relative error of $\delta^+_{DP}f(a)$ in approximating $f\'(a)$ -- log-log
pylab.xlabel('$h$')
pylab.ylabel('error')

pylab.legend(('$r$ - relative error',
              '$R$ - upper bound, $r$'))
```

Out[10]:
```
<matplotlib.legend.Legend at 0x1106832b0>
```



1.3.3 Analysis

Both of these graphs show that as h decreases and n increases, the error logarithmically approaches zero. However, after a certain point, the error begins to increase again (roughly around $n = 26, h = 10^{-8}$). At small values of h, the limitations in the floating point representation of numbers may introduce error at the evaluation of h itself, the evaluation of $a + h$ relative to a, or division by h.

1.3.4 Conclusion

These plots and calculations were a great demonstration of the pitfalls of doing complex arithmetic with floating point numbers, as the error plots show.

1.4 Part 3 - Propagation of Errors

This section covers the propagation of error when evaluating recurrence relations.

1.4.1 Background

Recurrence Relation for a Sequence of Values The sequence of values $\{V_j\}_{n=0}^{\infty}$ defined by the definite integrals

$$V_j = \int_0^1 e^{x-1} x^j dx j = 0, 1, 2, \ldots$$

satisfies the recurrence relation

$$V_j = 1 - jV_{j-1} j = 1, 2, \ldots$$

and the inequalities

$$0 < V_{j+1} < V_j < \frac{1}{j} j = 1, 2, \ldots$$

This can be show by the following:

$$V_0 = \int_0^1 e^{x-1} x^j dx \Rightarrow f(x) = x^j, g(x) = e^{x-1} f'(x) = jx^{j-1}, g'(x) = e^{x-1} \Rightarrow x^j e^{x-1} \Big|_0^1 - j \int_0^1 x^{j-1} e^{x-1} dx V_0 = \int_0^1 e^{x-1}$$

1.4.2 Implementation

Recurrence Relation There is really only one function being evaluated for this: the recurrence relation

$$V_j = 1 - jV_{j-1}$$

This is implemented with the function `recurrence_relation`, which takes as input a new coefficient j and the previously computed value V_{j-1}. Of course, the initial input V_0 must come from the evaluation of

$$V_0 = \int_0^1 e^{x-1} x^j dx = 1 - \frac{1}{e}$$

Naturally, for the calculations involving an added ϵ value, this initial calculation for V_0 is replaced by $V_0 + \epsilon$.

For the non-ϵ calculations, the outputs are saved in files `j.txt` (the j values), `v.txt` (the V_j values), and `e.txt` (the $\frac{V_j}{j!}$ values). The calculations including ϵ values share in j values, but output their own `v2_e{1..3}.txt` and `e2_e{1..3}.txt` files.

```
In [11]: %pylab inline
         pylab.rcParams['figure.figsize'] = (10,6)
         matplotlib.rcParams.update({'font.size': 16})
         matplotlib.rcParams.update({'axes.labelsize': 20})
         matplotlib.rcParams.update({'xtick.labelsize': 12})
         matplotlib.rcParams.update({'ytick.labelsize': 12})
         matplotlib.rcParams.update({
             'font.family': 'Helvetica, Arial, sans-serif'
         })

         %config InlineBackend.figure_format = 'retina'
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [12]: names = [ 'j', 'v', 'e', 'v2_e1', 'v2_e2', 'v2_e3', 'e2_e1', 'e2_e2', 'e2_e3' ]
         data = {name: loadtxt('../res/part3/'+name+'.txt') for name in names}
```

```
In [13]: from IPython.display import (HTML, display)

         table = [ data['j'], data['v'], data['e'] ]

         table[0] = np.hstack(('j', table[0]))
         table[1] = np.hstack(('V_j', table[1]))
         table[2] = np.hstack(('V_j / j!', table[2]))

         table = zip(*table)

         display(HTML('<h3>Initial Calculation, &epsilon; = 0.0<h3>'))
         display(HTML(
             '<table><tr>{}</tr></table>'.format(
                 '</tr><tr>'.join(
                     '<td>{}</td>'.format('</td><td>'.join(str(_) for _ in row)) for row in tabl
                 )
          ))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

As the table illustrates, as j increases, V_j approaches and then surpasses 0, that is until j becomes sufficiently large such that V_j starts to alternate between very large and very small quantities. This is likely due to the propagation of error as V_j gets further and further away from its initial state.

```
In [14]: from IPython.display import (HTML, display)

         table = [ data['j'], data['v2_e1'], data['e2_e1'] ]

         table[0] = np.hstack(('j', table[0]))
         table[1] = np.hstack(('V_j', table[1]))
         table[2] = np.hstack(('| V_j / j! |', table[2]))

         table = zip(*table)

         display(HTML('<h3>Second Calculation, &epsilon; = 0.13<h3>'))
         display(HTML(
             '<table><tr>{}</tr></table>'.format(
                 '</tr><tr>'.join(
                     '<td>{}</td>'.format('</td><td>'.join(str(_) for _ in row)) for row in tabl
                 )
          ))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
In [15]: from IPython.display import (HTML, display)

         table = [ data['j'], data['v2_e2'], data['e2_e2'] ]

         table[0] = np.hstack(('j', table[0]))
         table[1] = np.hstack(('V_j', table[1]))
         table[2] = np.hstack(('| V_j / j! |', table[2]))

         table = zip(*table)

         display(HTML('<h3>Third Calculation, &epsilon; = 0.0024<h3>'))
         display(HTML(
             '<table><tr>{}</tr></table>'.format(
                 '</tr><tr>'.join(
                     '<td>{}</td>'.format('</td><td>'.join(str(_) for _ in row)) for row in tabl
                 )
          ))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
In [16]: from IPython.display import (HTML, display)

         table = [ data['j'], data['v2_e3'], data['e2_e3'] ]

         table[0] = np.hstack(('j', table[0]))
         table[1] = np.hstack(('V_j', table[1]))
         table[2] = np.hstack(('| V_j / j! |', table[2]))

         table = zip(*table)

         display(HTML('<h3>Final Calculation, &epsilon; = 0.000035<h3>'))
         display(HTML(
             '<table><tr>{}</tr></table>'.format(
                 '</tr><tr>'.join(
                     '<td>{}</td>'.format('</td><td>'.join(str(_) for _ in row)) for row in tabl
                 )
          ))
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

For each of the calculations involving an ϵ value, $|\frac{V_j}{j!}|$ approaches that ϵ value as j increases (until j becomes sufficiently large, of course).

Derivation of Recurrence Relation $\hat{V}_j = V_j + (-1)^j j!\epsilon$

$\frac{\hat{V}_j}{j!} = \frac{V_j}{j!} + (-1)^j\epsilon$

$\approx (-1)^j\epsilon$

1.4.3 Conclusion

This is yet another example of how error propagates throughout arithmetic computations due to the limitations of floating point number representations, but this is especially apparent in this section due to the sheer scale of the error.