# Project 1 - Taylor Series and Floating-Point Error

September 15, 2016

Ben Maxey
MATH 3316

## 1 Approximation of a Function by Taylor Polynomials

### 1.1 Procedure

In this section we were tasked with approximating the function $f(x) = e^x$ using Taylor polynomials of degree 4, 8, and 12, then analyzing the error associated with each approximation. The approximation was to be carried out in the range $-1 \leq x \leq 1$, as the maximum error within this range should remain relatively small. In order to compute values of the Taylor polynomials, I needed a function that could efficiently evaluate polynomials. Thus, the first step in approximating $e^x$ was to implement Horner's algorithm, which evalutates polynomials using nested multiplication. This algorithm was written in a function in the file "nest.cpp":

```
In [ ]: #include "nest.h"

        //Evaluates a polynomial at x.  The coefficients of the polynomial
        //should be stored in a.
        double nest(Matrix& a, double x)
        {
                int n = a.Cols() - 1;
                double output = a(n);

                for (int i = n - 1; i >= 0; i--)
                        output = a(i) + (x * output);

                return output;
        }
```

As parameters, the nest function takes a matrix containing the coefficients of the polynomial to be evaluated and a value at which to evaluate the polynomial. Horner's algorithm runs in $O(n)$ time and is significantly faster than just plugging in values, which would run in $O(n^2)$ time. The next task was to create matrices containing the coefficients of each of the three Taylor polynomials. For $e^x$ at $x = 0$, the coefficients of the Taylor polynomial are $a_n = \frac{1}{n!}$. In "proj1_a.cpp," I wrote a standalone function, "exponentialTaylor" that generates coefficent matrices for a given degree:

```
In [ ]: //Returns the factorial of n.
        int factorial(int n)
        {
                if (n <= 0)
                        return 1;

                int x = n, output = n;
                while (x > 1)
                        output *= --x;

                return output;
        }

        //Returns a row vector with coefficients of the Taylor polynomial of the
        //specified degree for e^x at x = 0.
        Matrix exponentialTaylor(int degree)
        {
                Matrix output(1, degree);

                for (int i = 0; i < degree; i++)
                        output(i) = 1.0/(factorial(i));

                return output;
        }
```

Using this function, I was able to generate coefficient matrices for degrees 4, 8, and 12. To generate the actual approximations of $e^x$, I simply called the nest function on each of these coefficient matrices for a range of x values between -1 and 1. Generating the values for the approximations was accomplished in the function, "evaluate." Evaluate takes two matrices as inputs: the first is the range of values over which the polynomial will be evaluated, and the second is the coefficient matrix of the polynomial to be evaluated.

```
In [ ]: //Returns a matrix whose entries are the value of f(x) for all entries in x
        //where f is a row vector containing the coefficients of a power series.
        Matrix evaluate(Matrix& x, Matrix& f)
        {
                Matrix output(1, x.Cols());

                for (int i = 0; i < x.Cols(); i++)
                        output(i) = nest(f, x(i));

                return output;
        }
```

The last step is to calculate the error of each of the three approximations. To do this, I wrote a function that generates a matrix containing the absolute value of the differences between parallel entries of two input matrices. To generate the error matrices for each of the approximations, I called this function on each of the approximations with a matrix containing the true values of $e^x$.

I could have used the abs() function found in the cmath library to take the absolute value, but this function rounds the values that it returns. When I used abs(), several small values were rounded to 0, which is inaccurate.

```
In [ ]: //Returns a matrix whose entries are the absolute value of the difference
        //between the values of f and p.
        Matrix error(Matrix& f, Matrix& p)
        {
                Matrix output(1, f.Cols());

                for (int i = 0; i < f.Cols(); i++)
                {
                        if (f(i) - p(i) >= 0)
                                output(i) = f(i) - p(i);
                        else
                                output(i) = p(i) - f(i);
                }

                return output;
        }
```

The entire process is shown here in the main function of "proj1_a.cpp."

```
In [ ]: #include <iostream>

        #include "matrix.hpp"
        #include "nest.h"

        int factorial(int n);
        Matrix exponentialTaylor(int degree);
        Matrix evaluate(Matrix& x, Matrix& f);
        Matrix error(Matrix& f, Matrix& p);

        int main(int argc, char** argv)
        {
                //Initialize matrix z.
                Matrix z = Linspace(-1.0, 1.0, 1, 201);
                z.Write("z.txt");

                //Initialize Taylor polynomial coefficient matrices.
                Matrix coeff4 = exponentialTaylor(4);
                Matrix coeff8 = exponentialTaylor(8);
                Matrix coeff12 = exponentialTaylor(12);

                //Evaluate coefficient matrices for all values in z.
                Matrix p4 = evaluate(z, coeff4);
                Matrix p8 = evaluate(z, coeff8);
                Matrix p12 = evaluate(z, coeff12);
```

```
        //Write p4, p8, and p12.
        p4.Write("p4.txt");
        p8.Write("p8.txt");
        p12.Write("p12.txt");

        //Evaluate exp(x) for all values in z, then write f.
        Matrix f(1, z.Cols());
        for (int i = 0; i < z.Cols(); i++)
                f(i) = exp(z(i));
        f.Write("f.txt");

        //Compute error for each vector and write the error vectors.
        Matrix err4 = error(f, p4);
        Matrix err8 = error(f, p8);
        Matrix err12 = error(f, p12);
        err4.Write("err4.txt");
        err8.Write("err8.txt");
        err12.Write("err12.txt");
}
```

### 1.2 Results

```
In [2]: %pylab inline
```

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

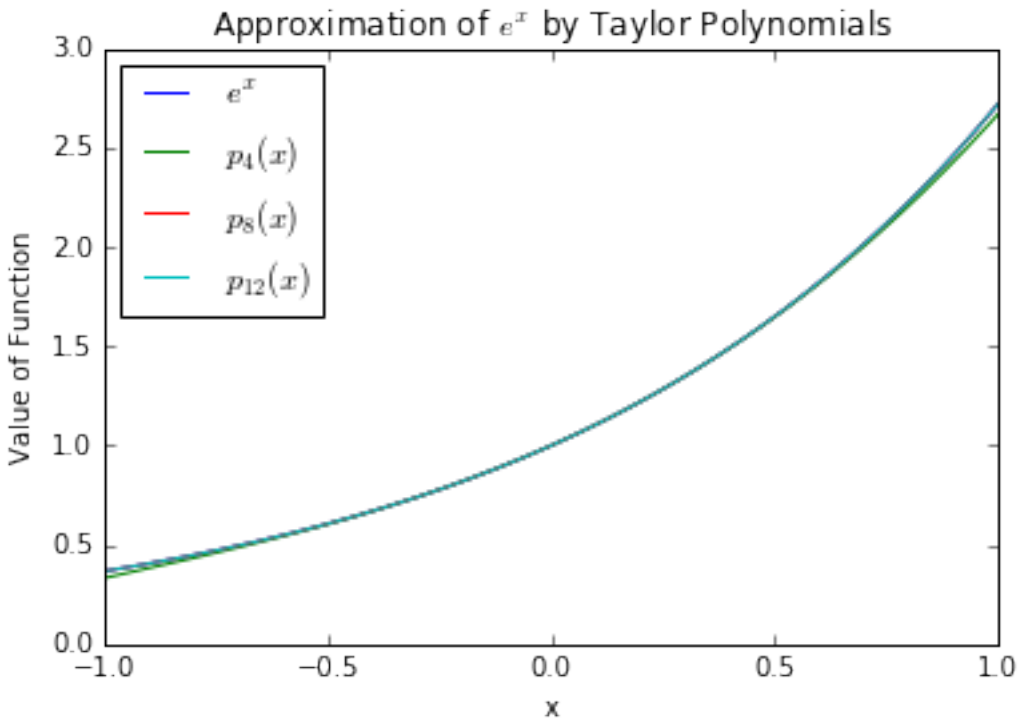
```
In [4]: z = loadtxt('z.txt')

        p4 = loadtxt('p4.txt')
        p8 = loadtxt('p8.txt')
        p12 = loadtxt('p12.txt')
        f = loadtxt('f.txt')

        err4 = loadtxt('err4.txt')
        err8 = loadtxt('err8.txt')
        err12 = loadtxt('err12.txt')
```

```
In [7]: plot(z,f,z,p4,z,p8,z,p12)
        xlabel('x')
        ylabel('Value of Function')
        title('Approximation of $e^x$ by Taylor Polynomials')
        legend(('$e^x$', '$p_4(x)$', '$p_8(x)$', '$p_{12}(x)$'), loc=2)
```

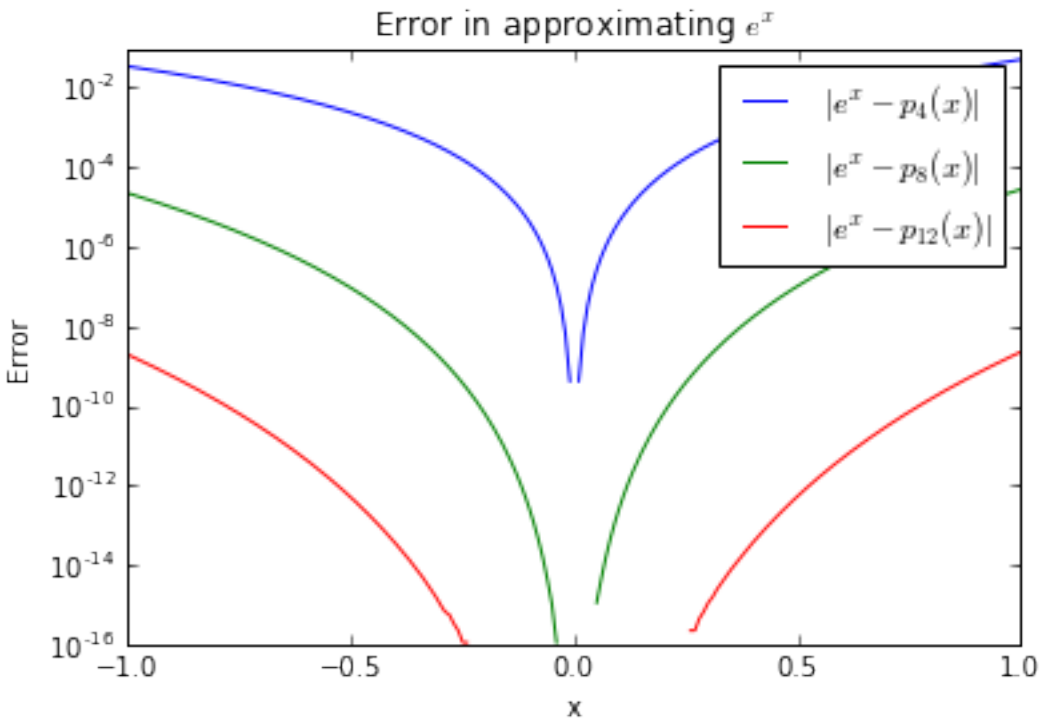
```
Out[7]: <matplotlib.legend.Legend at 0x7f350a550828>
```




From the above plot, it might be somewhat difficult to discern which of the approximations is the most accurate. In the interval $-1 \leq x \leq 1$, all approximations are effective, yet each tends to slightly underapproximate $e^x$. $p_{12}(x)$ is the best approximation, followed by $p_8(x)$, then finally $p_4(x)$. Because the differences between the approximations are so difficult to see when plotted normally, it is helpful to examine the error of each approximation.

```
In [8]: semilogy(z,err4,z,err8,z,err12)
        xlabel('x')
        ylabel('Error')
        title('Error in approximating $e^x$')
        legend(('$|e^x - p_4(x)|$', '$|e^x - p_8(x)|$', '$|e^x - p_{12}(x)|$'))

Out[8]: <matplotlib.legend.Legend at 0x7f350a5af668>
```




As expected, the error for $p_4(x)$ is much greater than the error for $p_8(x)$, which is much greater than the error for $p_{12}(x)$. For each approximation, the error decreases dramatically as x approaches the center of the approximation $(x = 0)$. Theoretically, the maximum error for each approximation should adhere to the following formula:

$$|e^x - p_n(x)| = |E_{n+1}|$$

$$|E_{n+1}(x)| = \frac{e^{c_x}}{(n+1)!}|x|^{n+1}, -1 < c_x < 1$$

$$|E_{n+1}(x)| \leq \frac{e^1}{(n+1)!}(1)^{n+1}$$

$$|E_{n+1}(x)| \leq \frac{e}{(n+1)!}$$

$$|e^x - p_4(x)| \leq \frac{e}{5!} < 0.02266$$

$$|e^x - p_8(x)| \leq \frac{e}{9!} < 7.4910 * 10^{-6}$$

$$|e^x - p_{12}(x)| \leq \frac{e}{13!} < 4.3654 * 10^{-10}$$

However, the errors obtained in this project rose above these upper bounds. The most likely explanation for this increase in error is that precision was lost due to rounding each time any calculation was made. Because there were so many operations required to calculate the Taylor polynomials, precision was lost several times.

## 2 Errors in a Forward Finite Difference Approximation

### 2.1 Procedure

In this section we were tasked with approximating the derivative of the function $f(x) = x^3$ at $x = 3$ using a forward finite difference approximation. This approximation is $\delta^+ f(a) = \frac{f(a+h)-f(a)}{h}$. As $h$, also called the increment, approaches 0, $\delta^+ f(a)$ approaches $f'(a)$. For this section I used this sequence of increments: $h = 2^{-n}(n = 1, 2, ..., 52)$. Both the increments and the sequence of 1 to 52 were stored in matrices named h and n, respectively. Next, I generated estimates of $f'(3)$ using the forward difference approximation for each increment in h. This was accomplished by simply plugging values into the equation given above. Then, I calculated the relative error in each of the approximations using the true value of $f'(3)$, which is -27. Finally, for each increment, I calculated the upper bound for the relative error. The maximum error is given by $R = |\frac{f''(a)}{2f'(a)}|h + |\frac{f(a)\epsilon_{DP}}{f'(a)}|\frac{1}{h}$. The entire "proj1_b.cpp" file is given below.

```
In [ ]: #include <iostream>
        #include <cmath>

        #include "matrix.hpp"

        Matrix estimate(Matrix& increments);
        Matrix getRelError(Matrix& estimates);
        Matrix getMaxError(Matrix& increments);

        int main()
        {
                Matrix n = Linspace(1, 52, 1, 52);
                n.Write("n.txt");

                Matrix h(1, 52);
                for (int i = 0; i < 52; i++)
                        h(i) = exp2(-n(i));
                h.Write("h.txt");

                Matrix estimates = estimate(h);
                estimates.Write("estimates.txt");

                Matrix r = getRelError(estimates);
                r.Write("r.txt");

                Matrix R = getMaxError(h);
                R.Write("R.txt");
        }

        //This function returns an estimate of f'(3) using the
        //forward difference estimate with given increments
        Matrix estimate(Matrix& increments)
        {
```

```
        Matrix output(1, 52);

        for (int i = 0; i < 52; i++)
        {
                double z = 3 + increments(i);
                double fah = pow(z,-3);
                output(i) = (fah - (1.0/27.0)) / increments(i);
        }

        return output;
}

//This function returns the relative error of the given estimates
//with respect to the true value of f'(3)
Matrix getRelError(Matrix& estimates)
{
        Matrix output(1, 52);
        double correct = -(1.0/27);

        for (int i = 0; i < 52; i++)
        {
                if (estimates(i) < correct)
                        output(i) = correct - estimates(i);
                else
                        output(i) = estimates(i) - correct;
        }

        return output;
}

//This function returns the upper bound on the relative error
//of an estimate for each of the given increments
Matrix getMaxError(Matrix& increments)
{
        Matrix output(1, 52);
        double c1 = (12.0 * pow(3.0,-5.0)) / (2.0 * (1.0/27.0));
        double c2 = (1.0/27.0) * pow(2,-52) / (1.0/27.0);

        for (int i = 0; i < 52; i++)
                output(i) = (c1 * increments(i)) + (c2 * 1.0/increments(i))

        return output;
}
```

## 2.2 Results

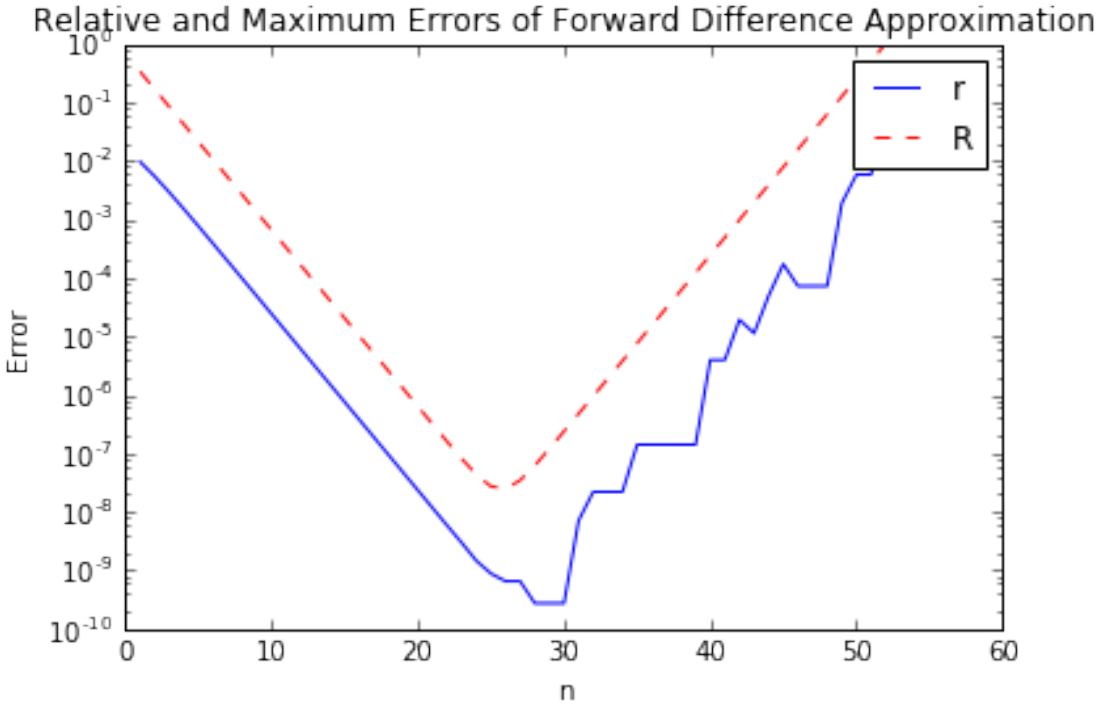
```
In [1]: %pylab inline
```

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

```
In [3]: n = loadtxt('n.txt')
        h = loadtxt('h.txt')
        r = loadtxt('r.txt')
        R = loadtxt('R.txt')
```

```
In [4]: semilogy(n, r, 'b-', label='r')
        semilogy(n, R, 'r--', label='R')
        xlabel('n')
        ylabel('Error')
        title('Relative and Maximum Errors of Forward Difference Approximation')
        legend()
```

```
Out[4]: <matplotlib.legend.Legend at 0x7fed29c50198>
```

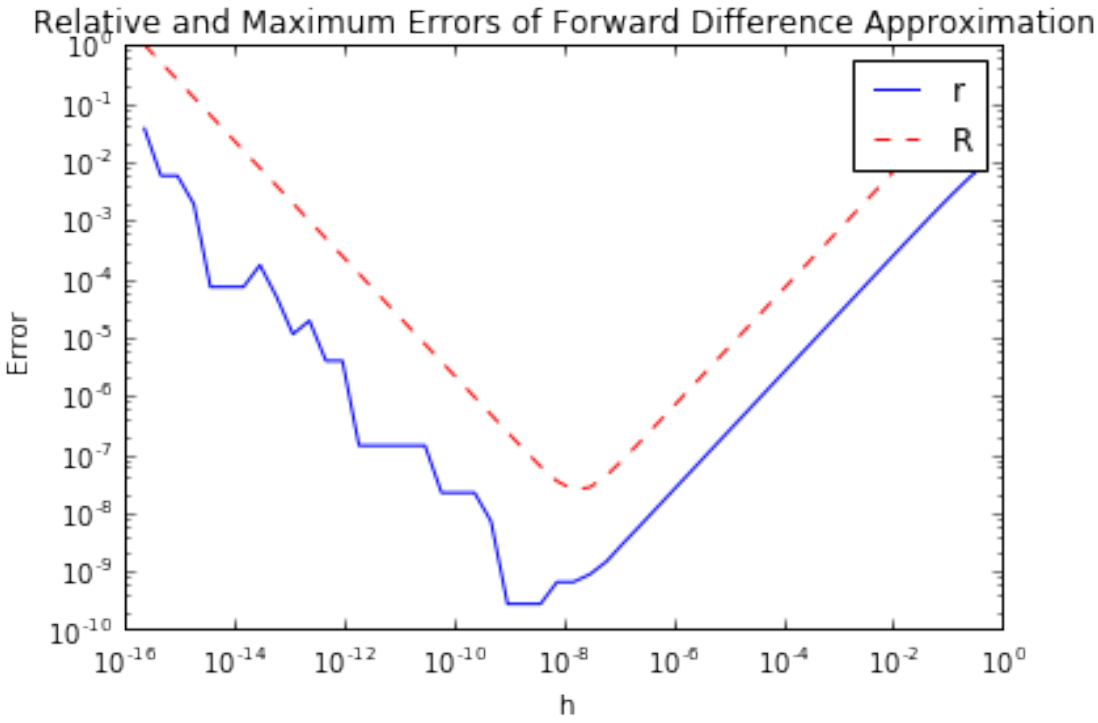



This plot shows the relative error for each n, and the upper bound for the error (R). As n approaches 30, the relative error and its upper bound decreases exponentially, but increases exponentially afterward. The approximation becomes more accurate as h grows smaller, but after a certain point the affects of catastrophic cancellation occur. As $f(a+h)$ and $f(a)$ become increasingly close, the inherent error in subtraction outweighs the accuracy of having a small increment.

```
In [5]: loglog(h, r, 'b-', label='r')
        loglog(h, R, 'r--', label='R')
```

```
xlabel('h')
ylabel('Error')
title('Relative and Maximum Errors of Forward Difference Approximation')
legend()
```

```
Out[5]: <matplotlib.legend.Legend at 0x7fed29383208>
```




This plot shows the same relative error and maximum relative error, but as a function of h instead of n. The best increment to use seems to be $2^{-30}$, as it has the lowest relative error The jagged line seen after $n = 30$ can also be attributed to the effects of catastrophic cancellation. Once it begins to affect the accuracy of the estimate, the error is affected non-logarithmically.

# 3 Appendix

## 3.1 Makefile

```
In [ ]: # %load Makefile
all: proj1_a proj1_b

proj1_a: proj1_a.o nest.o matrix.o
	g++ $^ -o $@

proj1_b: proj1_b.o matrix.o
	g++ $^ -o $@
```

```
proj1_a.o: proj1_a.cpp
	g++ -c proj1_a.cpp

proj1_b.o: proj1_b.cpp
	g++ -c proj1_b.cpp

nest.o: nest.cpp
	g++ -c nest.cpp

matrix.o: matrix.cpp
	g++ -c matrix.cpp

clean:
	rm -f *o
	rm -f *txt
```