Project 1 – Taylor Series and Floating-Point Error

Conrad Appel - 9/16/2014 - MATH 3316

Part 1:

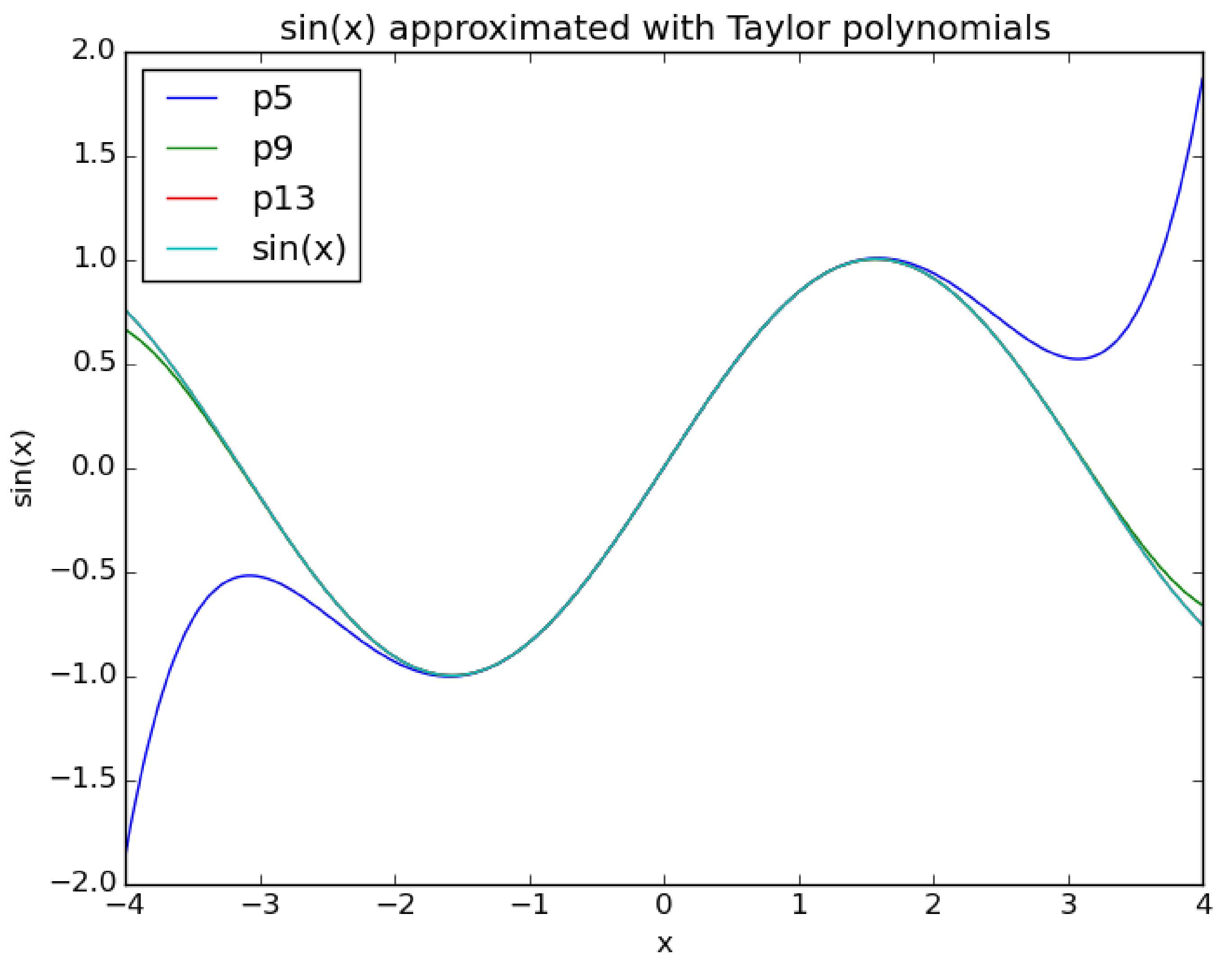
In the first part of this project, we needed to implement an algorithm for nested multiplication used to evaluate n-degree polynomials given to us by our book. After implementation, we tested the accuracy of our algorithm by computing a Taylor polynomial for `sin(x)`. My implementation of `double nest(Mat &a, double x);` is as follows (where `a` is a row vector of coefficients for the nested multiplication):

```
double nest (Mat &a, double x) {
   int n = a.Cols() - 1;
   double p = a(n);
   for(int i = n - 1; i >= 0; i--) {
      p = a(i) + (x * p);
   }
   return p;
}
```

For the vector of coefficients, I expanded the Taylor polynomial for sin(x) at 5, 9, and 13 powers as such:

```
gr0 + x - 0 - x^3/3! + 0 + x^5/5!
0 + x - 0 - x^3/3! + 0 + x^5/5! - 0 - x^7/7! + 0 + x^9/9!
0 + x - 0 - x^3/3! + 0 + x^5/5! - 0 - x^7/7! + 0 + x^9/9! - 0 - x^11/11! + 0 + x^13/13!
```

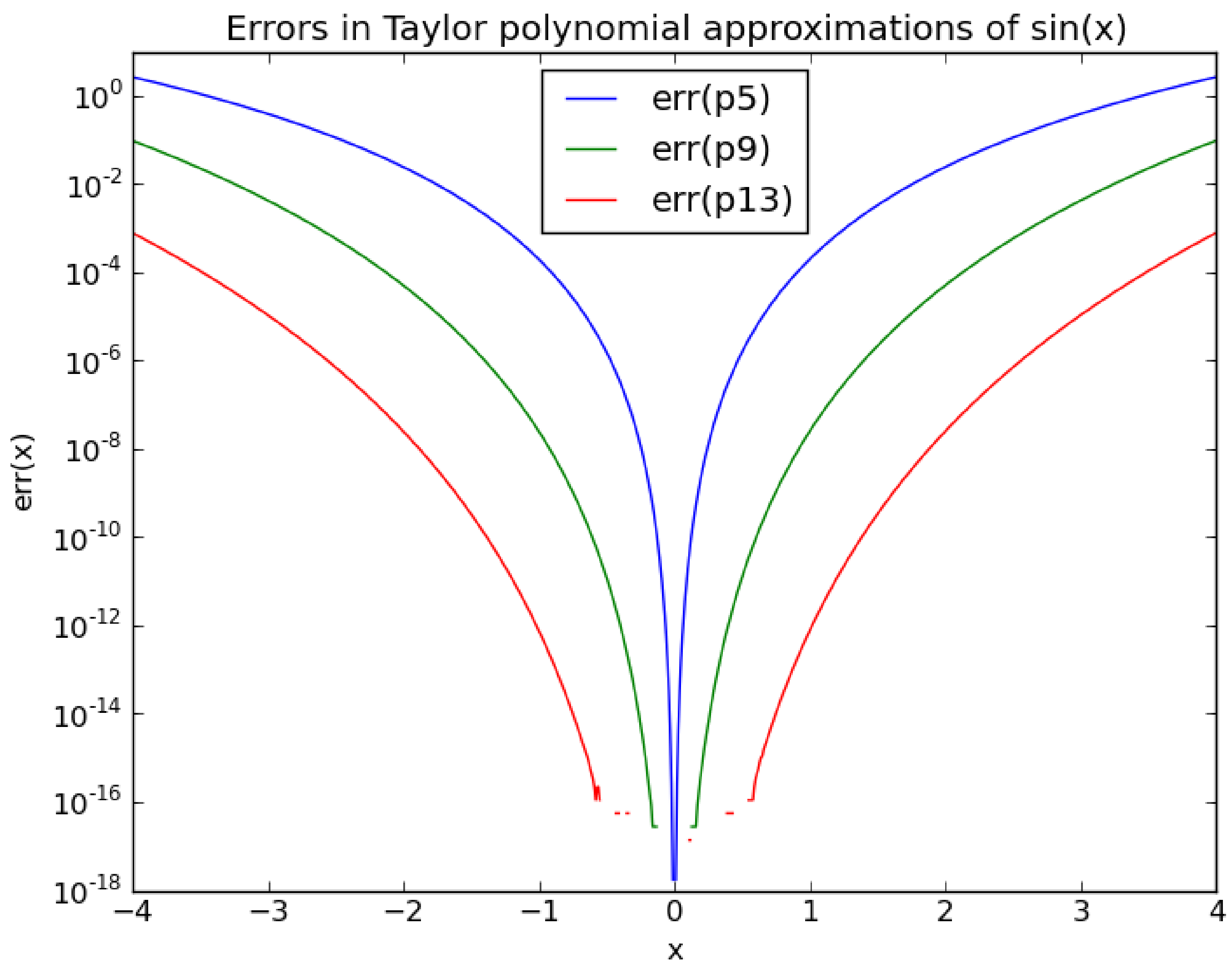
Then, for all values [-4, -3.99, -3.98, ..., 3.98, 3.99, 4.0] (801 in total) as `x`, used the nested multiplication function to calculate the approximation of the `sin(x)` function, saving the resulting numbers to files named `p5.txt`, `p9.txt`, and `p13.txt`. I then loaded all of the values, including the actual values of `sin(x)` into a Python script which allowed me to generate a graph for the approximations, making it easy to visualize how accurate the Taylor approximations were.



After computing and plotting all of the Taylor approximations at all 801 points with 5, 9, and 13 powers of Taylor polynomial, I calculated the error for each using the following formulas, which give the real differences between the actual value and our approximate value:

```
| sin(x) - p[5](x) |
| sin(x) - p[9](x) |
| sin(x) - p[13](x) |
```

With all of the calculated errors, I plotted the errors on one graph using Matplotlib's `semilogy()` function, which uses a logarithmic scale for the y-axis.



After viewing the results of the calculations of the Taylor polynomials and their corresponding errors on graphs, it was immediately obvious that the higher-powered Taylor polynomials were significantly more accurate than the lower ones. We know this because the graph of the degree-13 polynomial is much closer to the actual graph of `sin(x)`, as well as its error being much smaller on the error graph in comparison to the other two calculations. Assuming this pattern continues, I believe it is safe to assume that higher-degree Taylor polynomials will give us increasingly more accurate results.

Part 2:

For part two we used the backwards difference equation to estimate `f'(5)`, derived from function `f(a) = a^-2`, with increments of `h = 2^-n` where `n = [1, 2, 3, ..., 52]`. The equation used for the backwards difference is as such (where `a` = 5):

```
d-f(a) = [ f(a) - f(a - h) ] / h
```

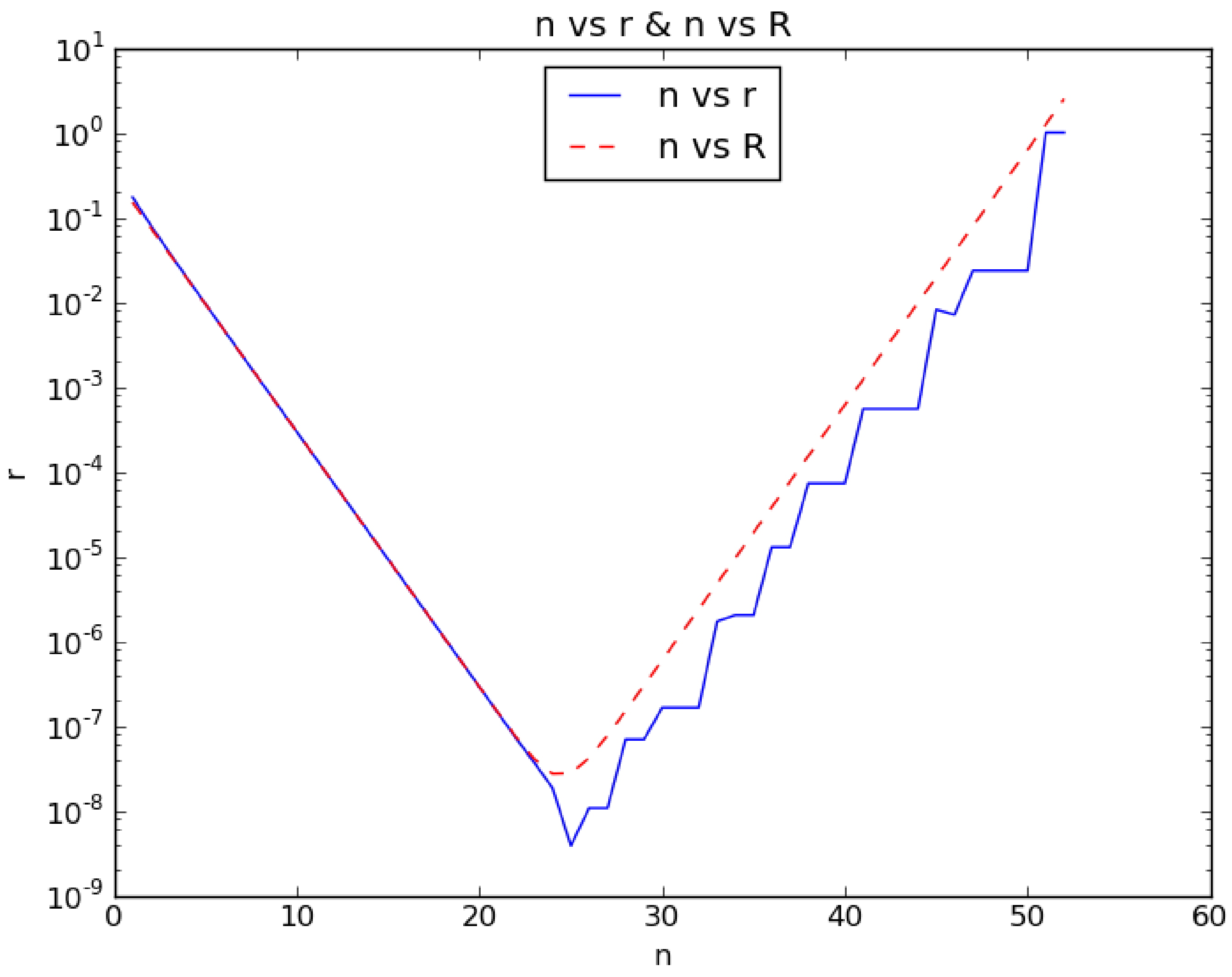
Because the reasoning behind this part of the lab is finding the relevant errors on a set of calculations, I used the results of the backwards difference equation as well as the actual, hand-calculated value of `f'(5)` to calculate both the upper bound (named capital "R") on the error, and the relative error (lowercase "r") associated with estimating the derivative using the backwards difference equation. The upper bound `R` was calculated using the following equation (where eDP (double-precision floating point roundoff) = 2^-52):

```
c1 = | f^(2)(a) / [2f^(1)(a)] |
c2 = | [f(a) * eDP] / f^(1)(a) |
R = (c1 * h) + (c2 * 1/h)
```

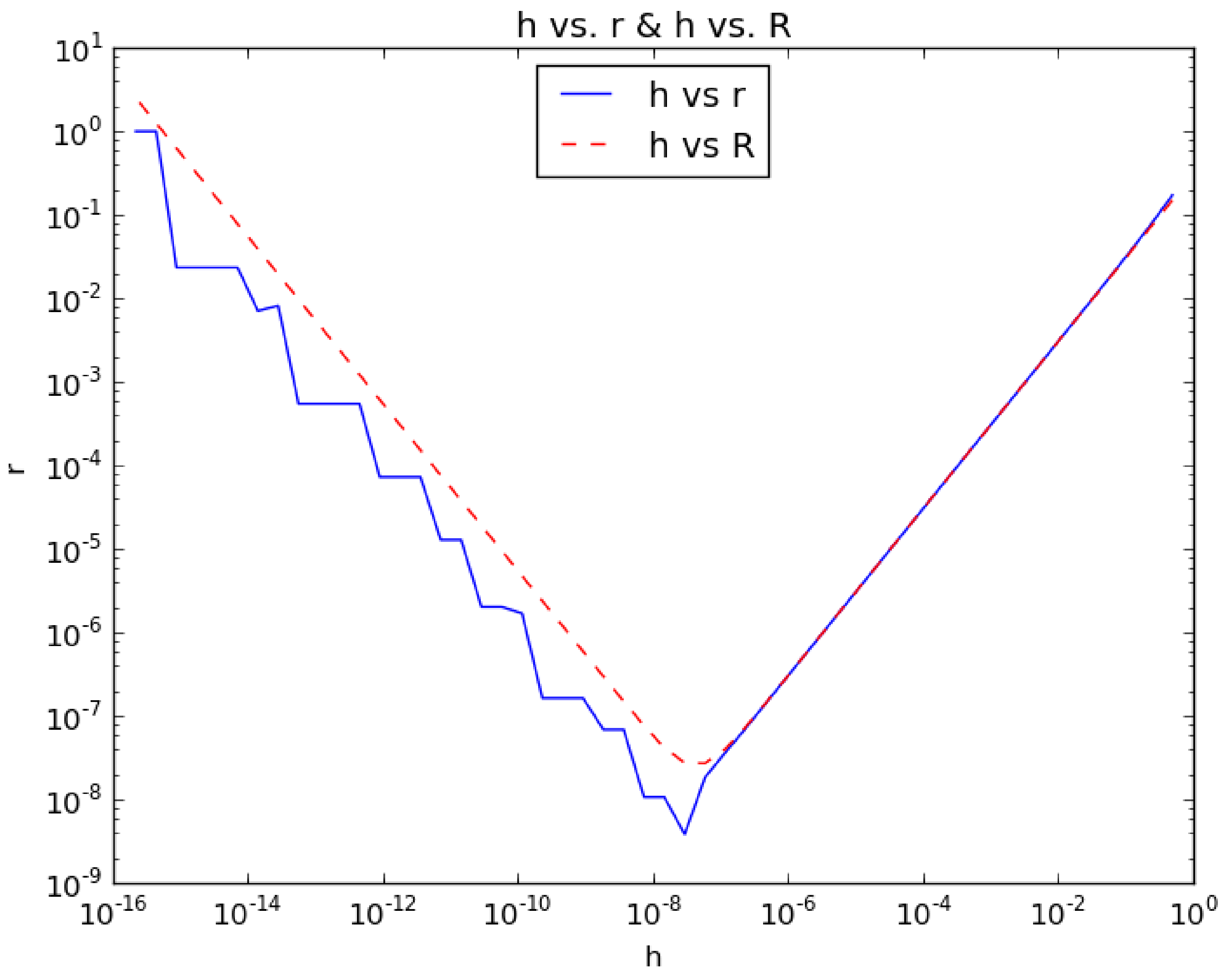
and relative error `r` is calculated with:

```
r = | f^(1)(5) - d-f(a) | / f^(1)(5)
```

Both errors are outputted to files, along with the increments `n` and the calculated `h` values (using `h = 2^-n`) for plotting usage in Matplotlib.



In this first plot, the `semilogy()` command is used for plotting, which uses a logarithmic scale for the y-axis. In this plot, the `n` values used as the domain are linear [1, 2, 3, ..., 52], while the `r` and `R` values follow a logarithmic pattern, which, when used with the `semilogy()` command, results in straight lines.



This second plot uses the `loglog()` command for plotting, which uses a logarithmic scale for both the x- and y-axes. This results in straight-trending lines because the `h` values used for the domain follow a pattern representative of a logarithmic function while the error values `r` and `R` are both logarithmic.

In my analysis of the two lines, I noticed a few important patterns. For one, the relative error always falls at or below the upper bound for the error, which is what should happen. A second observation is that the errors suddenly shoot up when `n` = 26, which is something I did not expect.

Looking more at why the error shot up at `n` = 26, I noticed the following:

- Up until `n` = 26, the value of the backwards difference approached .016, which is what I would expect, as .016 is the real value of `5^-2` .
- Right at `n` = 26, the value of the backwards difference jumped from about `.016000000` to about `.015999999` , which was the first step in the opposite direction of where the value should be going.
- I believe that this jump is caused by dividing by `h` in both `d-f(a)` and `R` . This error is caused by `h` becoming exponentially tinier, leading to the possibility of double-precision floating point number roundoff as well as increasingly large numbers being stored in the computer due to dividing numbers by such a small denominator.

Because of this large increase in error, it makes sense to use the value of `n` right before an increase in error occurs: at `n` = 25. This will give us the best approximation that we can get while still using this algorithm. The value of `d-f(5)` we get with `n` = 25 is `-0.0160000000614673`, which is *very* close to what the actual value of `f'(5)` is.