Scientific Computing Project 2

Jose Alarcon-Silva

November 2021

Introduction

In this project, we are interested in numerical methods for polynomial interpolation and function integration.

For the first part of the project, we consider the function

$$f(x) = \frac{1}{cx^2 + 1} \tag{1}$$

Where c is a real, positive number. We use various different methods of interpolation and apply them to the function $f(x)$ with a polynomial $p(x)$ of degree n using equidistal points. We use Newton's method and natural cubic spline interpolation.

In the second part of the project, we consider the integral

$$I_1 = \int_0^2 \sqrt{4 - x^2}dx \tag{2}$$

and

$$I_2 = \int_0^1 \frac{4}{1 + x^2}. \tag{3}$$

We write Python code to implement the composite trapezoid rule, composite Simpson's rule, and Gaussian quadrature to calculate the integrals I_1 and I_2.

Part 1

For this part of the project, we are interested in the function (1).

$$\frac{1}{cx^2 + 1}$$

Here, c is a real, positive number.

Using Python, we develop a divided difference table and implement Newton interpolation to find our interpolating polynomial of degree n.

We then use a natural cubic spline to interpolate the function.

To plot the function and our resulting polynomials after implementing Newton's method and natural cubic spline in Python code, we use the matplotlib in Python. To graph the function, we use numpy's linspace operator to generate 100 equidistal points in the interval $[-1, 1]$.

In the first part of this project, we use equidistal points x_i between -1 and 1 such that

$$x_i = \frac{2i}{n} - 1 \tag{4}$$

for $i = 0, ..., n$. We use this code to interpolate $f(x)$ with $n = 6, 10, 14, 18$ for the values of $c = 1, 4, 25$.

We would expect that the error increases as c increases and decrease as n increases. As c increases, we are generating smaller numbers and we may experience roundoff error. We also expect our polynomial to perfectly interpolate the data at the given nodes. This restriction of interpolating the data at the nodes puts further restrictions on the polynomial at points that are not at the nodes. As f is a smooth, continuous function, this should decrease the error between nodes.

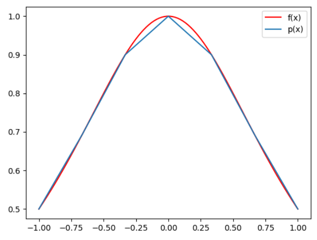

Figure 1: $c = 1, n = 6$.

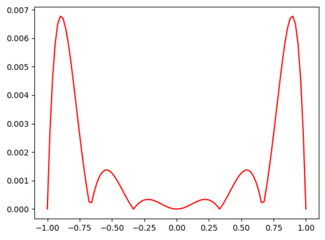

Figure 2: $c = 1, n = 6$ error.

As expected, at our nodes the error is zero but increase elsewhere. There is a significant increase near the ends of the intervals while it decreases near 0.

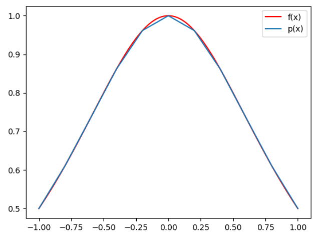

Figure 3: $c = 1, n = 10$.

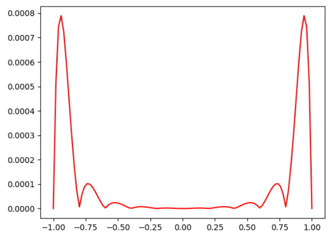

Figure 4: $c = 1, n = 10$ error.

As n increased and there are more points to interpolate, we can see that the graph of our polynomial is smoothing out near zero.

In comparison to $n = 6$, we can see that the error has decreased significantly.

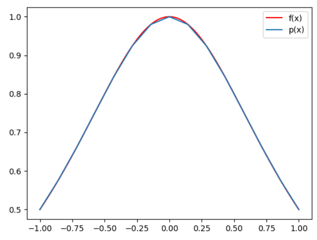

Figure 5: $c = 1, n = 14$.

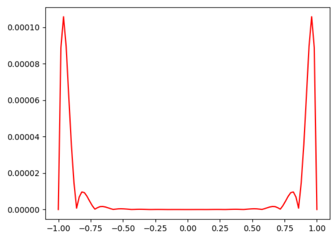

Figure 6: $c = 1, n = 14$ error.

As we increase n, we can see that the error is continuing to fall at every point.

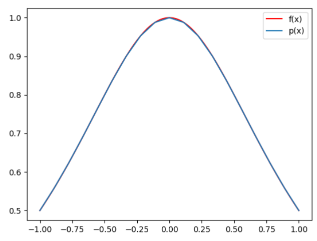

Figure 7: $c = 1, n = 18$.

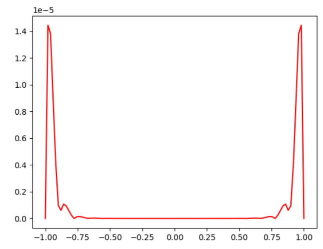

Figure 8: $c = 1, n = 18$ error.

Again, we see a decrease in the overall error.

Next, we consider $c = 4$. We use the same equation for x_i, as mentioned in equation (4).

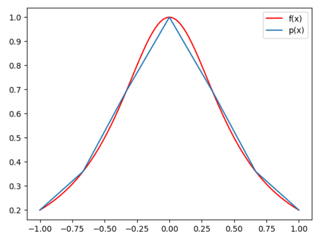

Figure 9: $c = 4, n = 6$.

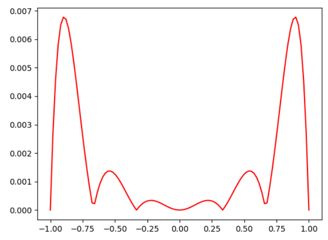

Figure 10: $c = 4, n = 6$ error.

Here, we can see that the error increased as our c increased while our n decreased. The error remains highest near the ends of the interval and significantly smaller near zero. Here, our error is as high as 0.007 near 1 and -1. Our error is almost exactly the same as it was for $c = 1, n = 6$.

By increasing n, we expect our error to decrease.

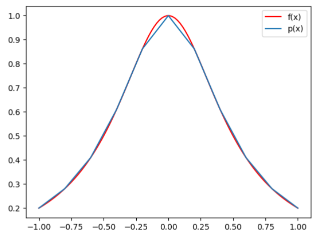

Figure 11: $c = 4, n = 10$.

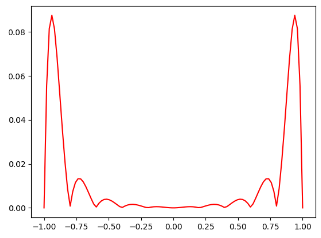

Figure 12: $c = 4, n = 10$ error.

However, our error increases. Indeed, near the ends of the interval we see that our error is as high as 0.08, in comparison to near 0.007 for $n = 6$. This may be due to the the fact that increasing c means we are generating very small numbers, and we may experience round-off error as we increase c.

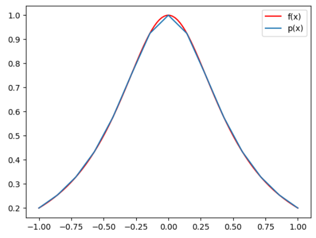

Figure 13: $c = 4, n = 14$.

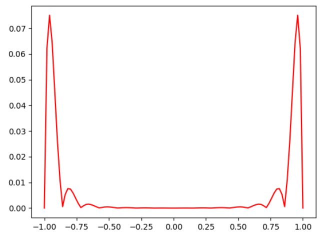

Figure 14: $c = 4, n = 14$ error.

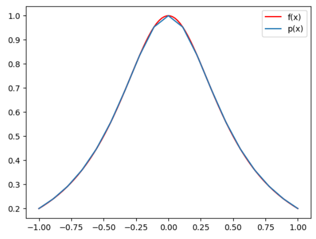

Figure 15: $c = 4, n = 18$.

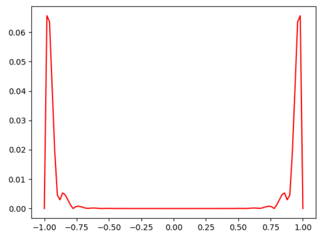

Figure 16: $c = 4, n = 18$ error.

We do find that the error decreases. However, it does not decease by nearly as much as we saw for $c = 1$. At $n = 18$ and $c = 4$, we find that we have error larger than 0.06, while for $c = 1$, we found that the error was significantly smaller near the end of the intervals. However, we can see that the error has smoothed out as one approaches zero, with very small errors in this region.

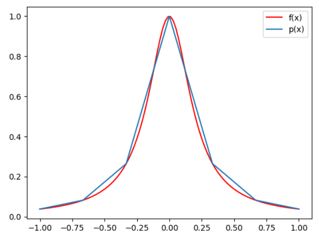

Figure 17: $c = 25, n = 6$.

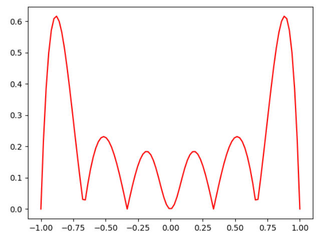

Figure 18: $c = 25, n = 6$ error.

We see that there is a very significant error in approximating around the ends of the interval. This is likely due to the fact that we are generating very small numbers with $c = 25$. In this case, increasing n may lead to an increase in error due to the fact that we are dividing by small numbers.

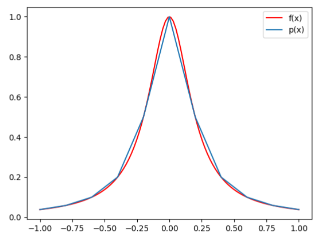

Figure 19: $c = 25, n = 10$.

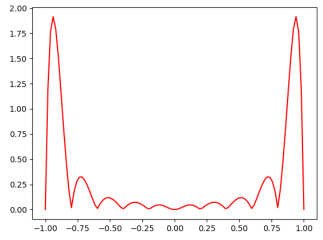

Figure 20: $c = 25, n = 10$ error.

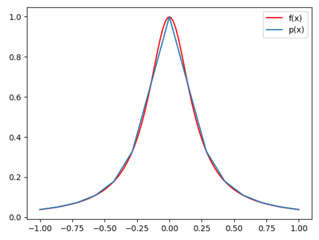

Figure 21: $c = 25, n = 14$.

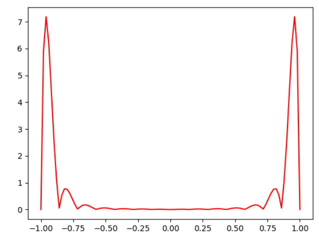

Figure 22: $c = 25, n = 14$ error.

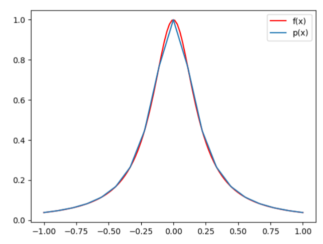

Figure 23: $c = 25, n = 18$.

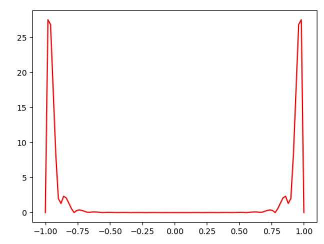

Figure 24: $c = 25, n = 18$ error.

As we can see, our error increased significantly as n increased at $c = 25$. For the next part, we consider the nodes

$$x_i = \cos(\frac{(2i-1)\pi}{2n} \tag{5}$$

Note that these nodes are not necessarily equidistal. In fact, we find that $x_0 = x_1$ if we were to use seven equally spaced points in the interval. Given this fact, we make some modifications to the code for Newton's method. First, we ensure that there are no repeated nodes. In doing so, the nodes are no longer equidistal. Secondly, the cosine function is not strictly increasing on the interval, so we must sort the list of nodes. After doing so, we may apply the same method as we did previously. We expect the error here to be larger as we may have nodes close to each other or very far apart, meaning, in a large interval we may not have any node at all. However, by increasing n we may expect error to decrease as more nodes are taken.

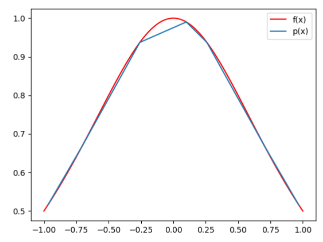

Figure 25: $c = 1, n = 6$.

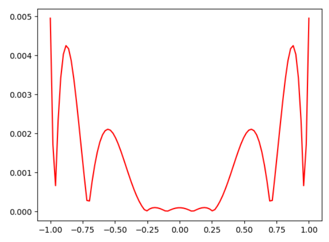

Figure 26: $c = 25, n = 6$ error.

As expected, we can see that there is a large gap where our polynomial fails to interpolate the function properly near zero as the nodes are not equally spaced. However, the overall error remains fairly small, with the error being largest near 1 and approximately 0.005.

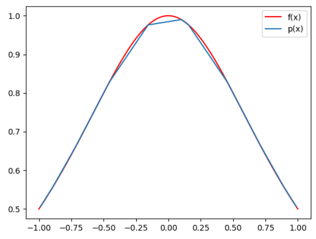

Figure 27: $c = 1, n = 10$.

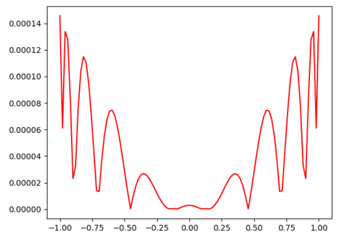

Figure 28: $c = 25, n = 10$ error.

As expected, we see the error decrease. As we are taking cosine over the nodes, we see that the error oscillates over the function, falling at each node and increasing away from the node.

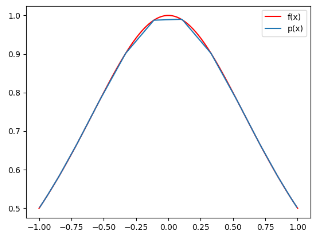

Figure 29: $c = 1, n = 14$.

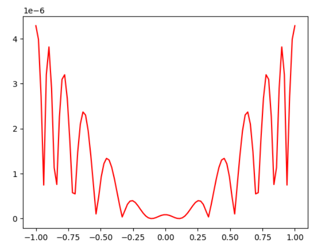

Figure 30: $c = 1, n = 14$ error.

Our error continues to fall as n increases and becomes very small, in the neighborhood of 1e-6.

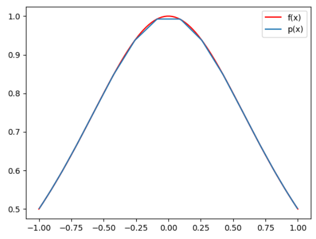

Figure 31: $c = 1, n = 18$.

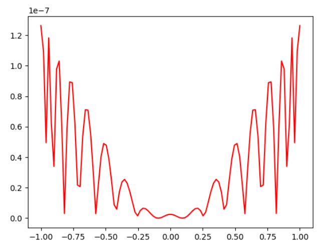

Figure 32: $c = 1, n = 18$ error.

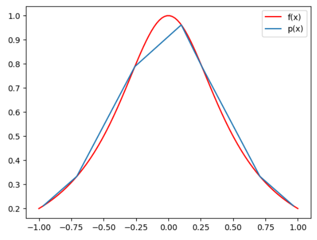

Figure 33: $c = 4, n = 6$.

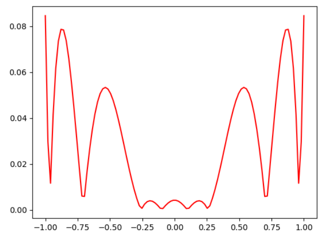

Figure 34: $c = 4, n = 6$ error.

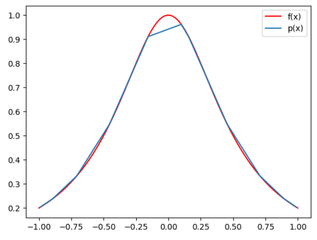

Figure 35: $c = 4, n = 10$.

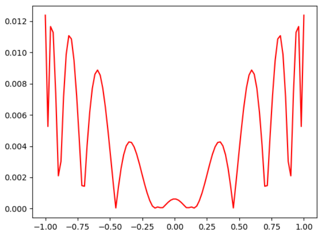

Figure 36: $c = 4, n = 10$ error.

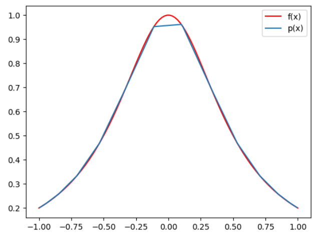

Figure 37: $c = 4, n = 14$.

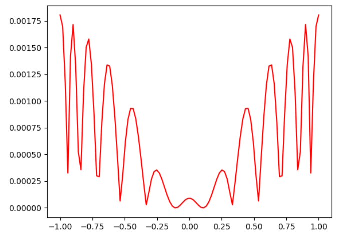

Figure 38: $c = 4, n = 14$ error.

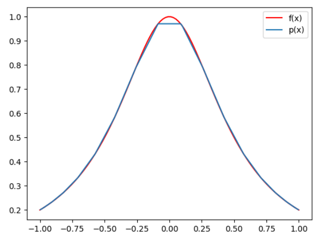

Figure 39: $c = 4, n = 18$.

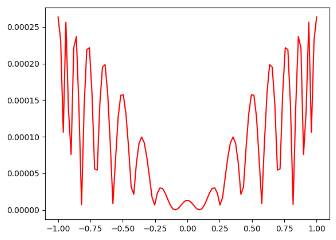

Figure 40: $c = 4, n = 18$ error.

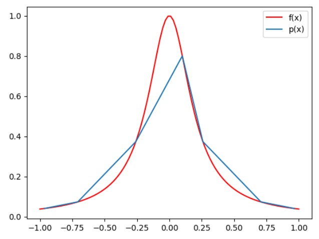

Figure 41: $c = 25, n = 6$.

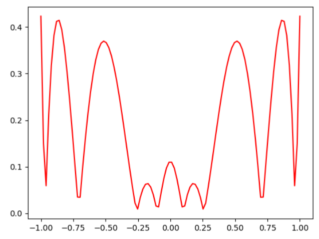

Figure 42: $c = 25, n = 6$ error.

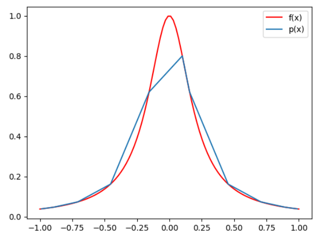

Figure 43: $c = 25, n = 10$.

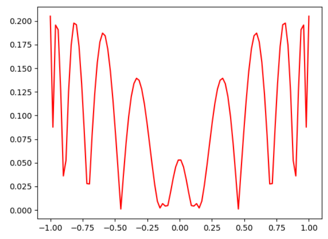

Figure 44: $c = 25, n = 10$ error.

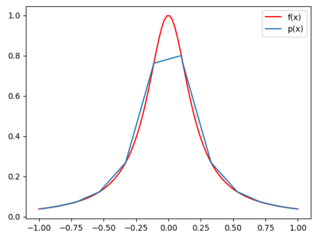

Figure 45: $c = 25, n = 14$.

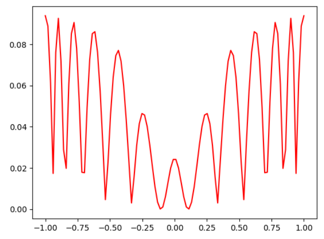

Figure 46: $c = 25, n = 14$ error.

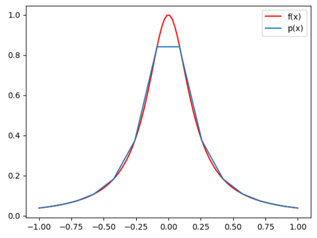

Figure 47: $c = 25, n = 18$.

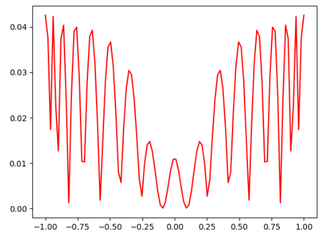

Figure 48: $c = 25, n = 18$ error.

As we can see from the above figures, using cosine rather than equisdistal points provides varying error. The error tends to oscillate, while we saw a comparatively smooth error for equidistal points.

Next, we use natural cubic spline interpolation. Spline functions consist of polynomial pieces on subintervals, which are then joined together in such a way that a continuous function results. We saw very small error using the cosine function to generate nodes. We expect the error to remain small, but similar regardless of c value as we are using cubic polynomials at each subinterval to approximate the function.

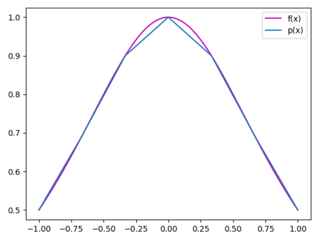

Figure 49: $c = 1, n = 6$.

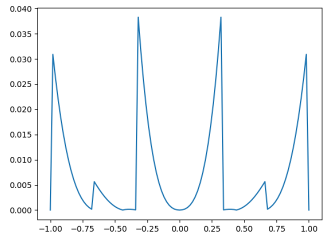

Figure 50: $c = 1, n = 6$ error.

We can see that the natural cubic spline interpolation does a reasonable job of interpolating the function given just $n = 6$. The error should fall as n increases, but not significantly.

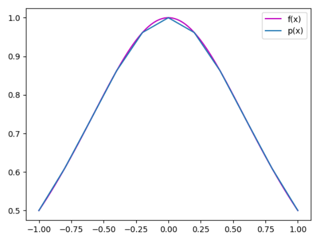

Figure 51: $c = 1, n = 10$.

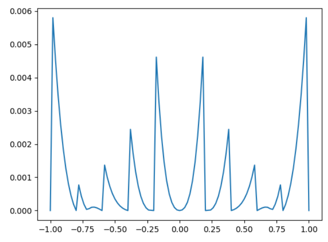

Figure 52: $c = 1, n = 10$ error.

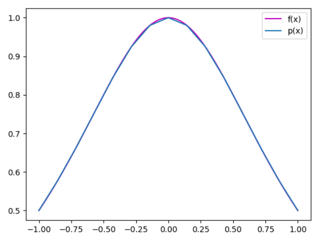

Figure 53: $c = 1, n = 14$.

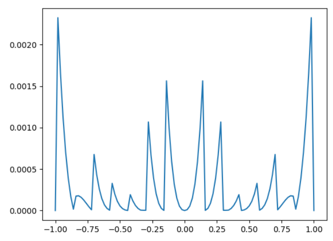

Figure 54: $c = 1, n = 14$ error.

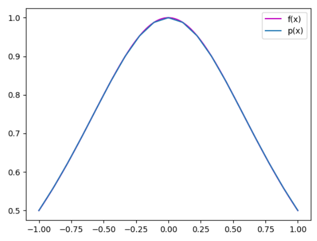

Figure 55: $c = 1, n = 18$.

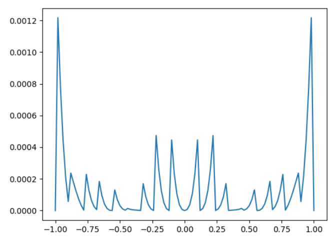

Figure 56: $c = 1, n = 18$ error.

Next, we let $c = 4$.

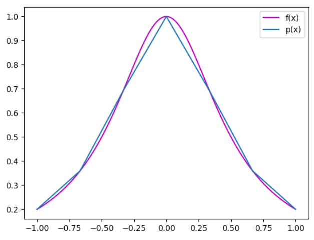

Figure 57: $c = 4, n = 6$.

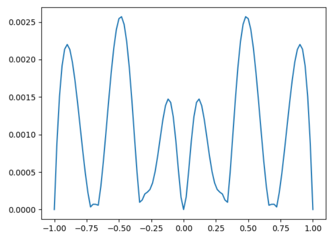

Figure 58: $c = 4, n = 6$ error.

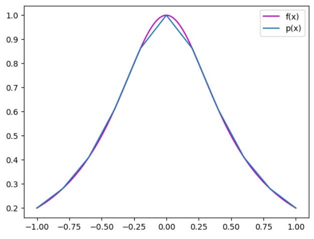

Figure 59: $c = 4, n = 10$.

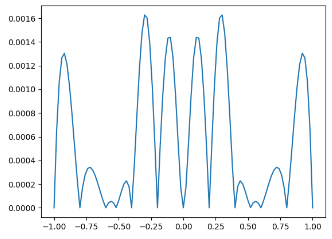

Figure 60: $c = 4, n = 10$ error.

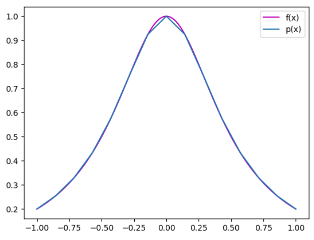

Figure 61: $c = 4, n = 14$.

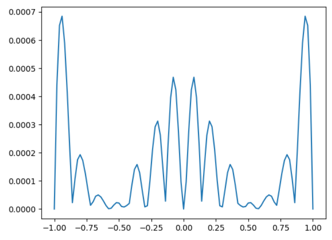

Figure 62: $c = 4, n = 14$ error.

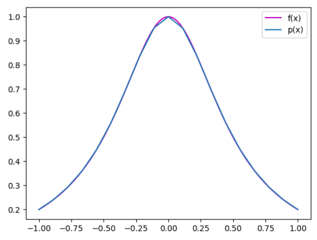

Figure 63: $c = 4, n = 18$.

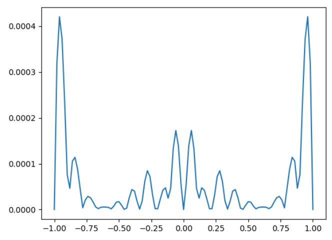

Figure 64: $c = 4, n = 18$ error.

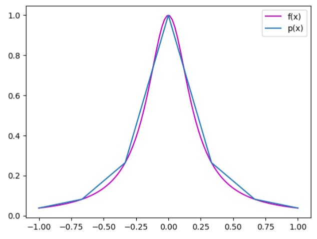

Figure 65: $c = 25, n = 6$.

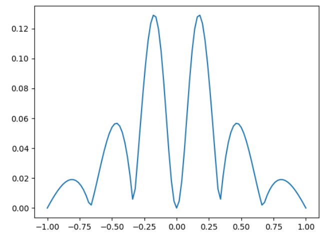

Figure 66: $c = 25, n = 6$ error.

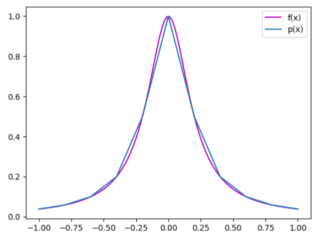

Figure 67: $c = 25, n = 10$.

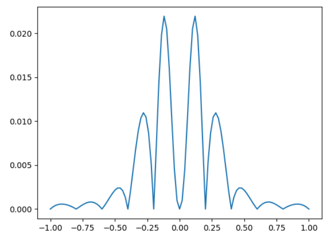

Figure 68: $c = 25, n = 10$ error.

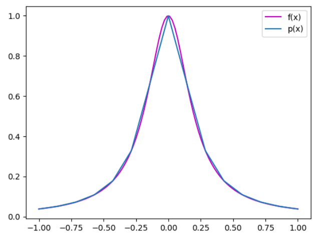

Figure 69: $c = 25, n = 14$.

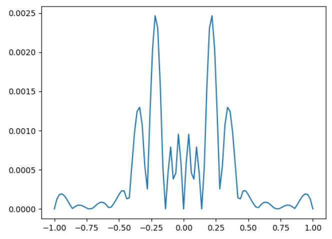

Figure 70: $c = 25, n = 14$ error.

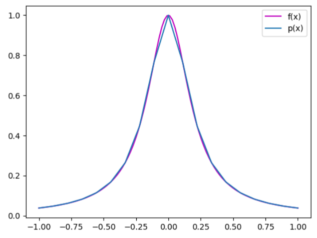

Figure 71: $c = 25, n = 18$.

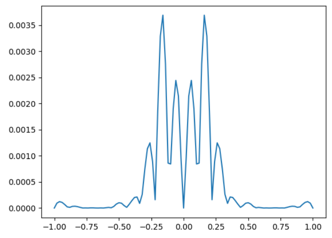

Figure 72: $c = 25, n = 18$ error.

Part II

In this part of the project, we use three different methods to estimate (2) and (3),

$$I_1 = \int_0^2 \sqrt{4 - x^2} dx$$

and

$$I_2 = \int_0^1 \frac{4}{1 + x^2} dx$$

We would expect I_2 to return more accurate results in general as I_1 includes the subtraction of nearby numbers, which may lead to significant error. I_2 avoids this problem entirely.

Of the three methods, we would expect Gaussian quadrature to have the smallest error, while Simpson's rule should have the highest error and the trapezoid rule should fall between the two.

In all three cases, as N increases, we expect the error to fall. N is taken to be $5, 10, 15, 20, 25, 30$ for the composite trapezoid rule and composite Simpson's rule, while $N = 2, 4, 8, 12, 16, 32$ for Gaussian quadrature.

We begin with the composite trapezoid rule:

$$\int_a^b f(x)dx \approx (h/2)[f(a) + 2\sum_{i=1}^{n-1} f(a+ih) + f(b)]$$

We write a program in Python to to estimate I_1 using the composite trapezoid rule.

We know,

$$I_1 = \int_0^2 \sqrt{4-x^2}dx = \pi$$

The error is calculated by using

$$|\pi - V|$$

Where V is our resulting value from the composite trapezoid rule. We use a similar formula for the composite Simpson's rule and Gaussian quadrature. The results are presented below.

N	N = 5	N = 10	N = 15	N = 20	N = 25	N = 30
Value	3.037048828883551	3.1045183262483182	3.1213914124755697	3.1284648797549814	3.132196310902997	3.1344431557571353
Error	0.10454382470624202	0.037074327341474866	0.02020124111422339	0.013127773834811673	0.009396342686796011	0.007149497832657836

The composite trapezoid rule applied to I_1 for $n = 5, 10, 15, 20, 25, 30$ and error estimate $|\pi - V|$.

We now turn to I_2. The error is calculated in the same way. The results are presented below.

N	N = 5	N = 10	N = 15	N = 20	N = 25	N = 30
Value	3.13492611381099	3.1399259889071587	3.1408519130232397	3.1411759869541287	3.141325986931254	3.1414074684073303
Error	0.006666539778803049	0.0016666646826344333	0.0007407405665533773	0.00041666663566441997	0.0002666666585389166	0.00018518518246279925

The composite trapezoid rule applied to I_2 for $n = 5, 10, 15, 20, 25, 30$ and error estimate $|\pi - V|$.

Next, we use a composite Simpson's rule to estimate the integrals I_1, I_2.

The composite Simpson's rule applies Simpson's rule to an even number of subintervals. Resulting in the formula,

$$\int_a^b f(x)dx = \frac{h}{3}[f(x_0) + 2\sum_{i=2}^{n/2} f(x_{2i-2}) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + f(x_n)]$$

We write Python code to carry out the numerical integration. Here, we note that $n/2$ may not be an integer, making summation for uneven n difficult. To remedy the error, we use the floor function in the math package of Python. However, this should significantly increase the error of the composite Simpson's rule.

We apply the formula to I_1. We find,

N	N = 5	N = 10	N = 15	N = 20	N = 25	N = 30
Value	1.3117822902541558	2.7345335483810715	2.761244790366831	2.9950617716980146	2.9623405106912553	3.0613637483768974
Error	1.8298103633356373	0.40705910520872157	0.380347863222962	0.14653088189177854	0.17925214289853786	0.08022890521289572

Composite Simpson's rule applied to I_1 for $n = 5, 10, 15, 20, 25, 30$ and error estimate $|\pi - V|$.

We now turn to I_2 and carry out similar calculations.

N	N = 5	N = 10	N = 15	N = 20	N = 25	N = 30
Value	1.4256410256410255	2.6843316877387924	2.6892080857208867	2.9277613802671523	2.8832289887466827	3.002185509089747
Error	1.7159516279487677	0.45726096585100073	0.45238456786890646	0.21383127332264085	0.25836366484311046	0.139407144500046

Composite Simpson's rule applied to I_2 for $n = 5, 10, 15, 20, 25, 30$ and error estimate $|\pi - V|$.

Here, we see that I_2 has a greater error than I_1

Next, we use Gaussian quadrature and apply it to our integral.

$$\int_{-1}^{1} f(x)dx = \sum_{i=1}^{n} w_i f(x_i)$$

The nodes and weights are given by the Legendre method.

Here, we need to generalize our formula to the integral of any closed interval, $[a, b]$. We do this by calculating,

$$y_i = \frac{b-a}{2} x_i + \frac{a+b}{2}$$

And our formula becomes,

$$\int_{a}^{b} f(x)dx = \frac{b-a}{2}(w_0 f(y_0) + ... + w_n f(y_n))$$

We use the Gauss-Legendre quadrature and take our x and w values from a table of values. I then imported this table of values into Python and implemented the Gaussian quadrature formula.

N	N = 5	N = 2	N = 4	N = 8	N = 12	N = 16, N = 32
Value	3.156072695039818	3.1451817756693496	3.1422775824170492	3.1418303356743915	3.1417015878568093	3.1416081552341395
Error	0.014480041450025016	0.003589122079556528	0.000684928827256126	0.00023768208459840068	0.00010893426701619191	1.5501644346382903e-05

Gaussian quadrature applied to I_1 for $N = 2, 4, 8, 12, 16, 32$ and error estimate $|\pi - V|$.

N	N = 5	N = 2	N = 4	N = 8	N = 12	N = 16, N = 32
Value	3.1410681399631675	3.141592639884753	3.1415926535882437	3.141592653589793	3.1417015878568093	3.1415926535897927
Error	0.0005245136266256445	1.3705040213807251e-08	1.5494272531668685e-12	4.440892098500626e-16	0.0	4.440892098500626e-16

Gaussian quadrature applied to I_2 for $N = 2, 4, 8, 12, 16, 32$ and error estimate $|\pi - V|$.

As we can see, Gaussian quadrature does, indeed, have the smallest error out of all three methods. This is due to the fact that Gaussian quadrature is exact for all polynomials of degree $\leq 2n + 1$.

Conclusion

In this project, we used various methods developed throughout the course to generate approximating polynomials and to do numeric integration.

For polynomial interpolation, we used a divided difference table to develop a Newton interpolating polynomial for our function and natural cubic spline interpolation.

In the first part of the project, we used equdistal points between -1 and 1 to interpolate our function (1). We expected that as c increased, we would see error increase between the function and our interpolating polynomial. In general, the statement held true as we were generating smaller and smaller numbers and the potential for roundoff error increased. We expected the error to fall as n increased. This too held in general. However, it was not necessarily true at very large values of c and n. This is due to the fact that more nodes places more restrictions on the polynomial, which may allow for very wide variations in the values that are not at the nodes.

Next, we opted to use the cosine function to generate points in the interval $[-1, 1]$. We found that, using this method allowed for the lowest error while natural cubic splines fell between the two methods in terms of overall error.

In the second part of the project, we estimated two integrals using three different methods. In general, we expected that I_2 would have a smaller error than I_1 in general as I_1 included the subtraction of nearby numbers, which may have increased the error. In terms of our methods, we expected Gaussian quadrature to provide for the best estimate as it is exact for polynomials of degree $\leq 2n + 1$. We found that Gaussian quadrature did, in fact, have the lowest error of the three methods, while Simpson's rule had the highest error as we used the floor function in Python to calculate the summations and the trapezoid rule fell between the two.