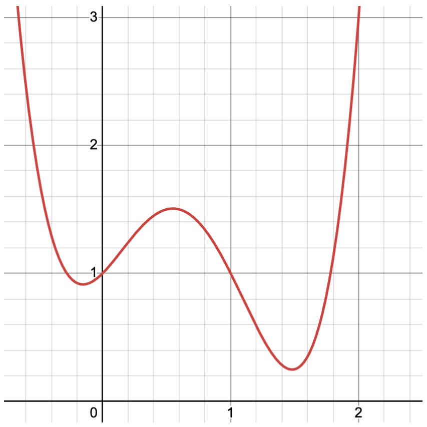
Numerical Differentiation and Integration

Mark Brautigam

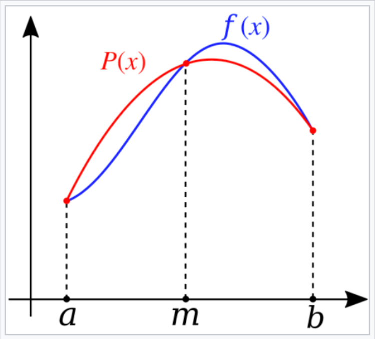
Math 370 • Numerical Analysis • 11 May 2025

Simpson's Rule



The idea behind Simpson's Rule is that we can approximate a cubic function with a composition of interpolated quadratic polynomials. In the function on the left, we can approximate the curve between $x = 0$ and $x = 1$ with a parabola opening downward, and the portion between $x = 1$ and $x = 2$ with a parabola opening upward.

We create the quadratic polynomial by letting it take the same values as the function at the endpoints and also at the midpoint. The image on the right (Wikipedia) shows such an interpolation.



Simpson's rule can be derived by approximating the integrand $f(x)$ (in blue) by the quadratic interpolant $P(x)$ (in red).

Wikipedia

The formula for one interval of the 1/3 or standard Simpson's rule is:

$$\int_a^b f(x)\,dx = \frac{1}{3}h[f(a) + 4f(a+h) + f(b)]$$

… where $h = \frac{b-a}{n}$ is the step size for $n = 2$.

To do this over several intervals, we need to have n be a multiple of 2, so we can divide each interval in half to utilize its midpoint in the calculations. Since we'll do this over a series of intervals, $f(b)$ for one interval ends up being the $f(a)$ for the next interval. So, the calculations at the edges of the intervals get counted twice, but the very first and very last interval have a value that is counted only once. The formula for an interval composed of an even number of subintervals shows that some calculations have a weight of 4 and others have a weight of 2, with the two endpoints having a weight of 1:

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

To do this calculation, we need to evaluate the function at every interval and its midpoints. If we have n intervals, we'll have $2n + 1$ calculations. Then we can just sum the calculations using the formula above. In Python, we could do this using a loop or an array with fancy indexing.

Simpson vs Trapezoid

In general, Simpson's method gives a closer approximation and it also converges more quickly. There may be a few weird situations where the trapezoid method gives better results. If the function is mostly linear over the interval of interest, the trapezoid method may have a closer approximation because it is using straight lines instead of quadratic curves. When evaluating discrete points, Simpson may be prohibited if the number of points is not odd (yielding an even number of intervals with midpoints).

> "Simpson's Rule relies on the function's fourth derivative to estimate the error, while the Trapezoidal Rule relies on the second derivative. If the fourth derivative of the function is significantly larger than the second derivative, the error in Simpson's Rule might be larger than the error in the Trapezoidal Rule, even if Simpson's Rule typically has a higher order of accuracy." (Google AI and Math Overflow)

https://mathoverflow.net/questions/483595/why-exactly-is-simpsons-rule-better-than-the-trapezoidal-rule

A real world example of a situation where we have discrete points might be a space mission. We may have a record of locations of a moon or Mars vehicle, a shuttle mission, or a satellite being launched or scrubbed. If the number of discrete points is not odd, it might be difficult to apply Simpson's method, and we might have to use the trapezoid method instead.

Finite Difference Approximation for Second Derivative

Finite Differences, Second Derivative, using $f(x), f(x-h), f(x-2h)$

Taylor series approximation for f(x-2h):

$$T_{-2} = f(x) + f'(x)(-2h) + f''(x)(-2h)^2\left(\frac{1}{2}\right) + f'''(x)(-2h)^3\left(\frac{1}{6}\right) + \cdots$$

$$T_{-1} = f(x) + f'(x)(-h) + f''(x)(-h)^2\left(\frac{1}{2}\right) + f'''(x)(-h)^3\left(\frac{1}{6}\right) + \cdots$$

Find coefficients such that

$$aT_{-2} + bT_{-1} + cf(x) = f''(x) + R_N$$
$$af(x) + bf(x) + cf(x) = 0$$
$$\boldsymbol{a + b + c = 0}$$

$$af'(x)(-2h) + bf'(x)(-h) = 0$$
$$-2ha - hb = 0$$
$$\boldsymbol{2a + b = 0}$$

$$a\left[f''(x)(-2h)^2\left(\frac{1}{2}\right)\right] + b\left[f''(x)(-h)^2\left(\frac{1}{2}\right)\right] = f''(x)$$
$$\boldsymbol{4a + b = \frac{2}{h^2}}$$

Solving this system of 3 equations, we get:

$$a = \frac{1}{h^2}, \qquad b = -\frac{2}{h^2}, \qquad c = \frac{1}{h^2}$$

Substituting back into the original equations to find the remainder (degree 3), we see

$$R \le a\left[f'''(x)(-2h)^3\left(\frac{1}{6}\right)\right] + b\left[f'''(x)(-h)^3\left(\frac{1}{6}\right)\right]$$
$$= f'''(x)\left(\frac{1}{6}\right)[(-8h^3)a + (-h)^3 b]$$
$$= f'''(x)\left(\frac{1}{6}\right)\left[(-8h^3)\left(\frac{1}{h^2}\right) + (-h^3)\left(-\frac{2}{h^2}\right)\right]$$
$$= f'''(x)(\frac{1}{6})(-\frac{6h^3}{\mathrm{h}^2})$$
$$= f'''(x)h$$

So $R < Mh$ where $M = max_{[a,b]}f'''(x)$

This looks different from some of our other error bounds, but it makes sense, because the error term is the third derivative, and we are evaluating the second derivative, so it makes sense that it depends linearly on the step size h.

Error Bound

The error bound would be

$$R \leq Mh$$

… where $M = max_{[a,b]} f'''(x)$ and h is the step size.

(I have no idea where the previous erroneous calculation came from.)

The best way to decrease the error is to decrease the step size h. None of the other variables can really be controlled.

Looping vs Indexing

In general, evaluating the formulas using a loop is equivalent to doing so using an indexed array. Here are some programming and performance tradeoffs to consider.

Some languages, such as C++, do not allow array operations. So, any attempt at using an indexed approach (such as with a library that provides such capability) would actually be using a loop behind the scenes, with no performance improvement.

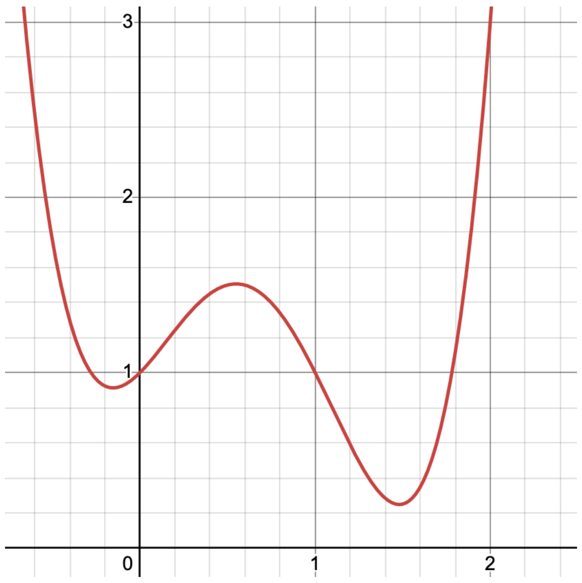
If the language natively provides array operations, such as Python does, then the provided array operations are likely to be faster than the equivalent loop variations.

Some languages deal with large arrays better than others. When using Python, use the numpy library for dealing with very large arrays. Fortran is also well known for dealing with very large arrays.

Array indexing can be fraught with indexing errors due to complicated combinations or programmer inexperience. But the same could be said about loop indexing, so this probably is not an important distinction.

(These programming opinions are basically my own based on my experience as a programmer. I did look up some questions about Python's ability to handle large arrays vs. other languages. Mostly I just found lots of arguments about programming styles and languages and not much real content, but I came away convinced that Python is well capable of handling very large data sets. I also didn't bring up MATLAB because I had only a small exposure to it 30 years ago.)

Python Coding



I used this function to test all the integrals and derivatives in the Python code. I wanted a function that oscillates a bit up and down. The function is

$$f(x) = 2x^4 - 5x^3 + 2x^2 + x + 1$$

I used these derivatives to calculate the actual values and error bounds:

$$f'(x) = 8x^3 - 15x^2 + 4x + 1$$

$$f''(x) = 24x^2 - 30x + 4$$

$$f'''(x) = 48x - 30$$

$$f^{(4)}(x) = 48$$

I used this integral to calculate the actual value and error bounds:

$$\int f(x)dx = \frac{2}{5}x^5 - \frac{5}{4}x^4 + \frac{2}{3}x^3 + \frac{1}{2}x^2 + x$$

On problem 3 my Simpson had approximations that exceed the error bounds, and the order of convergence was wrong. I ran Simpson in Excel to check it, and got a slightly different answer. I added a bunch of debugging code, and the Simpson code is now producing accurate results. But I didn't change any existing code, just added debugging lines. I removed the debugging code, and all is well. My only guess is that Colab was caching my code and not running new code after I changed things.

For problem 7, my OOC was probably not correct using grid sizes of 10 and 20 for comparison. For first derivative, 1.88. For second derivative, 3.23. I think these should be almost exactly 2. I changed to grid sizes of 40 and 80 and I got expected OOC of 1.94+ (~2) for both first and second derivatives.

Resources

https://en.wikipedia.org/wiki/Simpson%27s_rule

https://mathoverflow.net/questions/483595/why-exactly-is-simpsons-rule-better-than-the-trapezoidal-rule