Assignment 2
Numerical Integration

Prof. Darrell D. E. Long
CSE 13S – Winter 2022

Due: January 19th at 11:59 pm

1 Introduction

Mathematics knows no races or geographic boundaries;
for mathematics, the cultural world is one country.

—David Hilbert

When you took Calculus[1] you were presented with some simple and some complicated integrals, but regardless of their complexity, all of them had a *closed form*. What that means is that you can find an exact solution *analytically*. For example,

$$\int 3x^3 - 2x^2 + x - 1\,dx = \frac{3x^4}{4} - \frac{2x^3}{3} + \frac{x^2}{2} - x + c$$

where c is the constant of integration that arises when we calculate the *antiderivative*. Unfortunately, there are integrals, many of them which appear simple, that have no closed form.

For example, we might think $\int \sin(x^2)\,dx$ and $\int \cos(x^2)\,dx$ might be simple like $\int \sin(x)\,dx = -\cos(x)$ and $\int \cos(x)\,dx = \sin(x)$, but we would be mistaken. In fact, they are related to the *Fresnel integrals*

$$S(x) = \int_0^u \sin(\frac{1}{2}\pi x^2) dx \quad \text{and} \quad C(x) = \int_0^u \cos(\frac{1}{2}\pi x^2) dx.$$

The Fresnel integrals do not have closed forms, and so we are left with

$$\int \sin(x^2) dx = \sqrt{\frac{\pi}{2}} S\left(\sqrt{\frac{2}{\pi}} x\right) \quad \text{and} \quad \int \cos(x^2) dx = \sqrt{\frac{\pi}{2}} C\left(\sqrt{\frac{2}{\pi}} x\right).$$

In view of these examples, and many others, what are we to do? The answer is to use *numerical integration*.

2 Numerical Integration

[1] If you have forgotten (or never taken) calculus, do not despair. Go to a laboratory section for review: the concepts required for this assignment are just derivatives and the concept of integrals. You do not need to solve the integrals analytically.

B. Riemann

A Riemann sum (Bernhard Riemann, 17 September 1826–20 July 1866) is an approximation of a definite integral by a finite sum. The sum is calculated by partitioning the region into shapes (rectangles, trapezoids, parabolas, or cubics) that together form a region that is similar to the region being measured, then calculating the area for each of these shapes, and finally summing these small areas. This approach can be used to find a numerical approximation for a definite integral even if the fundamental theorem of calculus does not make it easy to find a closed-form solution.

Because the region filled by the small shapes is usually not exactly the same shape as the region being measured, the Riemann sum will differ from the area being measured. This error can be reduced by dividing up the region more finely, using smaller and smaller shapes. As the shapes decrease in size, the sum approaches the value of the integral.

The left Riemann sum approximates f by its value at the left-end point gives multiple rectangles with base Δx and height $f(a+i\Delta x)$. Doing this for $i = 0, 1, \ldots, n-1$, and summing the resulting areas gives

$$A_{\text{left}} = \Delta x \left[f(a) + f(a+\Delta x) + f(a+2\Delta x) + \cdots + f(b-\Delta x) \right].$$

The left Riemann sum overestimates if f is *monotonically decreasing* on this interval, and underestimates if it is *monotonically increasing*.

The right Riemann sum approximates f by its value at the right end-point. This gives multiple rectangles with base Δx and height $f(a+i\Delta x)$. Doing this for $i = 1, \ldots, n$, and summing the resulting areas produces

$$A_{\text{right}} = \Delta x \left[f(a+\Delta x) + f(a+2\Delta x) + \cdots + f(b) \right].$$

The right Riemann sum underestimates if f is monotonically decreasing, and overestimates if it is monotonically increasing.

The *midpoint rule* approximates f at the midpoint of intervals, giving $f(a+\frac{\Delta x}{2})$ for the first interval, for the next one $f(a+3\frac{\Delta x}{2})$, and so on until $f(b-\frac{\Delta x}{2})$. Summing up the areas gives

$$A_{\text{mid}} = \Delta x \left[f\left(a+\tfrac{\Delta x}{2}\right) + f\left(a+\tfrac{3\Delta x}{2}\right) + \cdots + f\left(b-\tfrac{\Delta x}{2}\right) \right].$$

Refer to Figures 1 and 2 for example plots of Riemann sums.

2.1 The Trapezoidal Rule

The *trapezoidal rule* (also known as the *trapezoid rule*) is an example of a closed Newton-Cotes quadrature. In numerical analysis, this is a common technique for approximating a *definite integral*.

It is assumed that the value of a function f defined on $[a,b]$ is known at $n+1$ equally spaced points: $a \le x_0 < x_1 < \cdots < x_n \le b$. It is a form of *closed* Newton–Cotes quadrature where $x_0 = a$ and $x_n = b$, using the function values at the interval end-points. Newton–Cotes formulas using $n+1$ points can be defined as

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i\, f(x_i)$$

where $x_i = a + ih$, with $h = (b-a)/n$.

The number h is called *step size*, w_i are called *weights*. The weights can be computed as the integral of Lagrange basis polynomials, but you do not need to worry about those here. Just know that they depend only on x_i and not on the function f.

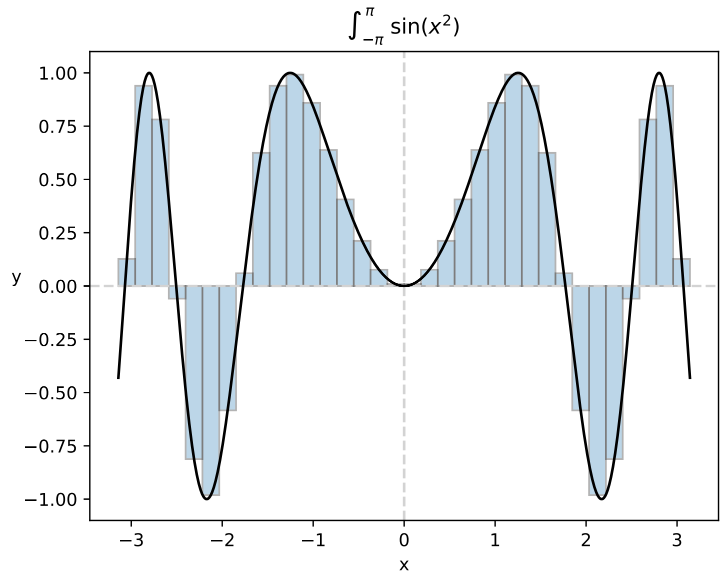

Figure 1: Midpoint Riemann sum for $\sin(x^2)$ over the range $[-\pi,\pi]$ with 50 partitions.

The trapezoidal rule works by approximating the region under the graph of the function $f(x)$ as a trapezoid and calculating its area. It follows that

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\big(f(a)+f(b)\big).$$

The trapezoidal rule may be viewed as the result obtained by averaging the left and right Riemann sums. The integral can be even better approximated by partitioning the integration interval, applying the trapezoidal rule to each subinterval, and summing the results. In practice, this *composite* trapezoidal rule is usually what is meant by "integrating with the trapezoidal rule." Let $\{x_k\}$ be a partition of $[a,b]$ such that $a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$ and Δx_k be the length of the k^{th} subinterval (that is, $\Delta x_k = x_k - x_{k-1}$), then

$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{n} \frac{f(x_{k-1})+f(x_k)}{2}\Delta x_k.$$

When the partition has a regular spacing, when all the Δx_k have the same value Δx, the formula can be simplified for calculation efficiency by factoring $h = \Delta x$ out:

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{h}{2}\big(f(x_0)+2f(x_1)+2f(x_2)+2f(x_3)+2f(x_4)+\cdots+2f(x_{n-1})+f(x_n)\big)\\ &= \frac{h}{2}\left(f(x_0)+2\sum_{i=1}^{n-1} f(x_i)+f(x_n)\right).\end{aligned}$$

As n increases, h decreases, and as the resolution of the partition increases the approximation becomes more accurate.

If we assume a fixed size $h = (b-b)/n$, then it is simple to write the code for the trapezoid rule.

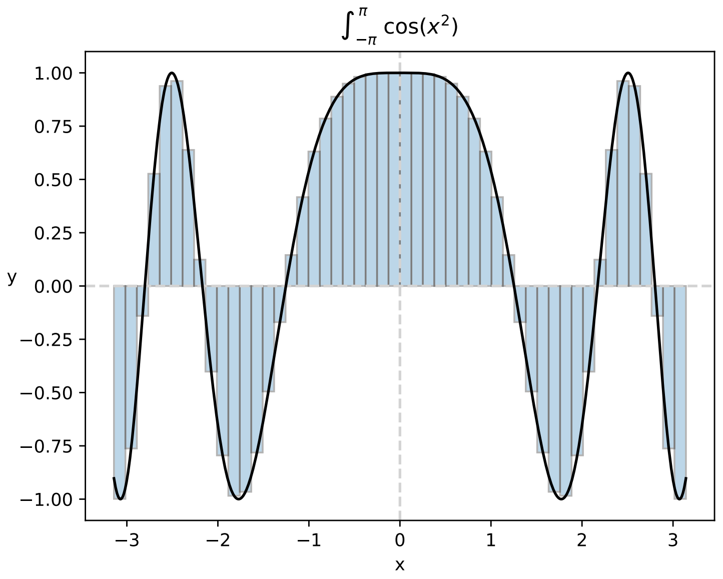

Figure 2: Midpoint Riemann sum for $\cos(x^2)$ over the range $[-\pi, \pi]$ with 50 partitions.

Implementation of the trapezoidal rule

```
def trapezoid(f, a, b, n):
    h = (b - a) / n
    sum = f(a) + f(b)
    for j in range(1, n):
        sum += 2.0 * f(a + j * h)
    sum *= h / 2.0
    return sum
```

2.2 Simpson's Rules

We can do better than using the simple trapezoidal rule by adopting one of Simpson's Rules for approximating definite integrals. Thomas Simpson (1710–1761) was a self-taught mathematician who supported himself during his early years as a weaver. His primary interest was probability theory, although in 1750 he published a two-volume calculus book entitled *The Doctrine and Application of Fluxions*. In German and some other languages, the simplest of these rules is named after Johannes Kepler, who derived it in 1615 after seeing it used for wine barrels (*Keplersche Fassregel*). The approximate equality in the rule becomes exact if f is a polynomial up to 3rd degree.

H. Simpson

Interpolation with polynomials evaluated at equally spaced points in $[a, b]$ yields a Newton–Cotes formulas, of which the *rectangle rule* and the *trapezoidal rule* are examples. Simpson's rule, which is based on a polynomial of order 2, is also a Newton–Cotes formula.

2.2.1 Simpson's 1/3 Rule

Simpson's 1/3 rule is also an instance of a Newton-Cotes quadrature formula. If the interval of integration $[a,b]$ is in some sense *small*, then Simpson's rule with $n=2$ subintervals will provide an adequate approximation to the exact integral. By small we mean that the function being integrated is relatively smooth over the interval $[a,b]$. For such a function, a smooth quadratic interpolant like the one used in Simpson's rule will give good results:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right].$$

2.2.2 Composite Simpson's 1/3 Rule

It is often the case that the function we are trying to integrate is not smooth over the interval. Typically, this means that either the function is highly oscillatory or lacks derivatives at certain points. One common way of handling this problem is by breaking up the interval $[a,b]$ into $n>2$ small subintervals. Simpson's rule is then applied to each subinterval, with the results being summed to produce an approximation for the integral over the entire interval. This approach is termed the *Composite Simpson's 1/3 rule.*

Suppose that the interval $[a,b]$ is split up into n sub-intervals, with n an even number. Then, the composite Simpson's 1/3 rule is given by

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

where $x_j=a+jh$ for $j=0,1,\ldots,n-1,n$ with $h=(b-a)/n$; in particular, $x_0=a$ and $x_n=b$. The error when using the composite Simpson's 1/3 rule is

$$-\frac{h^4}{180}(b-a)f^{(4)}(\xi),$$

where ξ is some number between a and b, and $h=(b-a)/n$ is the step length.

2.3 High Order Formulæ

These are not the only closed Newton-Cotes quadratures used for numerical integration. For example, there is the composite Simpson's 3/8 rule:

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{3h}{8}[f(x_0)+3f(x_1)+3f(x_2)+2f(x_3)+3f(x_4)+3f(x_5)+2f(x_6)+\\ &\qquad \cdots+3f(x_{n-2})+3f(x_{n-1})+f(x_n)]\\ &= \frac{3h}{8}\left[f(x_0)+3\sum_{i\neq 3k}^{n-1}f(x_i)+2\sum_{j=1}^{n/3-1}f(x_{3j})+f(x_n)\right] \quad \text{for } k\in\mathbb{N}_0.\end{aligned}$$

It has a smaller error term of

$$-\frac{h^4}{80}(b-a)f^{(4)}(\xi),$$

but can only be used this if n is a *multiple of three*.

Simpson's 3/8 rule is more difficult to implement directly, since it is not obvious how we write a loop for the first summation. Instead, we ask what that summation actually means: it means sum for all indices that are not multiples of 3 and the second summation for multiples of 3. The code then follows easily.

Implementation of composite Simpson's 3/8 rule

```python
def simpson_38(f, a, b, n):
    h = (b - a) / n
    sum = f(a) + f(b)
    for i in range(1, n):
        if i % 3 != 0:
            sum += 3 * f(a + i * h)
        else:
            sum += 2 * f(a + i * h)
    sum *= h * 3 / 8
    return sum
```

And, finally, there is the composite Boole's rule. Yes, that George Boole (2 November 1815–8 December 1864), the founder of Boolean logic. Boole's rule is also a Newton-Cotes formula with a decreased error term. In order to use Boole's composite rule, the number of partitions should be a multiple of 12.

$$\int_a^b f(x)\,dx = \frac{2h}{45}\left(7(f(x_0)+f(x_n)) + 32\left(\sum_{i\in\{1,3,\ldots,n-1\}} f(x_i)\right) + 12\left(\sum_{i\in\{2,6,\ldots,n-2\}} f(x_i)\right) + 14\left(\sum_{i\in\{4,8,\ldots,n-4\}} f(x_i)\right)\right)$$

that has the error term

$$-\frac{8}{945}h^7 f^{(6)}(\xi).$$

The formula for Boole's rule might appear daunting, but the code is not too bad. It just requires a little thought. Again, you ask, "What does each summation do?"

Implementation of Boole's rule

```python
def boole(f, a, b, n):
    h = (b - a) / n
    sum = 7 * (f(a) + f(b))
    for i in range(1, n, 2):     sum += 32 * f(a + i * h)
    for i in range(2, n - 1, 4): sum += 12 * f(a + i * h)
    for i in range(4, n - 3, 4): sum += 14 * f(a + i * h)
    sum *= h * 2 / 45
    return sum
```

If you are intrigued by these and other numerical computations, then we encourage you to do some reading on you own, or take a course in the Applied Mathematics department.

- Burden, Richard L., and J. Douglas Faires. *Numerical Analysis*, 7th edition, 2001. Thomson Learning ISBN 0-534-38216-9.
- Abramowitz, Milton, and Irene A. Stegun, *eds. Handbook of mathematical functions with formulas, graphs, and mathematical tables.* Vol. 55. US Government printing office, 1970.

3 Taylor Series

Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

—Donald Knuth

As we know, computers are simple machines that carry out a sequence of very simple steps, albeit very quickly. Unless you have a special-purpose processor, a computer can only compute *addition*, *subtraction*, *multiplication*, and *division*. If you think about it, you will see that the functions that might interest you when dealing with real or complex numbers can be built up from those four operations. We use many of these functions in nearly every program that we write, so we ought to understand how they are created.

If you recall from your calculus class, with some conditions a function $f(x)$ can be represented by its Taylor series (Brook Taylor, 18 August 1685–29 December 1731) expansion near some point a:

$$f(x) = f(a) + \sum_{k=1}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

Note: when you see Σ with definite limits, you should generally think of a `for` loop.

Since we cannot compute an infinite series, we must be content to calculate a finite number of terms. In general, the more terms that we compute, the more accurate our approximation.

For example, if we expand to 10 terms we get:

$$\begin{aligned} f(x) = f(a) &+ \frac{f^{(1)}(a)}{1!}(x-a)^1 + \frac{f^{(2)}(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \frac{f^{(4)}(a)}{4!}(x-a)^4 \\ &+ \frac{f^{(5)}(a)}{5!}(x-a)^5 + \frac{f^{(6)}(a)}{6!}(x-a)^6 + \frac{f^{(7)}(a)}{7!}(x-a)^7 + \frac{f^{(8)}(a)}{8!}(x-a)^8 \\ &+ \frac{f^{(9)}(a)}{9!}(x-a)^9 + \mathrm{O}((x-a)^{10}). \end{aligned}$$

In the case $a = 0$, then it is called a *Maclaurin series*. Often we choose 0 since it is simpler, but the closer to the value of x the better we will approximate the function. Note: $k! = k(k-1)(k-2) \times \ldots \times 1$, and by definition, $0! = 1$.

What is the $\mathrm{O}\left((x-a)^{10}\right)$ term? That is the *error term* that is "on the order of" the value in parentheses. This is different from the *big-O* that we will discuss with regard to algorithm analysis.

The number e, also known as *Euler's number* (Leonhard Euler, 1707–1783), is an irrational mathematical constant approximately equal to 2.71828, that appears pervasively in the natural and mathematical worlds. It is the base of the natural logarithm, it is the limit of $\lim_{n\to\infty}(1+\frac{1}{n})^n$ which was discovered by Jacob Bernoulli (6 January 1655–16 August 1705) in his work on the calculation of compound interest.

The function $f(x) = e^x$ is a very attractive function, since $f^{(k)}(x) = f^{(k-1)}(x) = \cdots = f'(x) = f(x) = e^x$. This is one of the simplest series when centered at 0, since $e^0 = 1$.

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \frac{x^8}{8!} + \frac{x^9}{9!} + \ldots$$

4 e^x

We have a nice series for e^x that converges quickly. In Figure 3, we use our expansion to 10 terms and plot for $e^x, x = 0, \ldots, 10$. We see that the approximation starts to diverge significantly around $x = 7$. What this tells us is that 10 terms are insufficient for an accurate approximation, and more terms are needed.

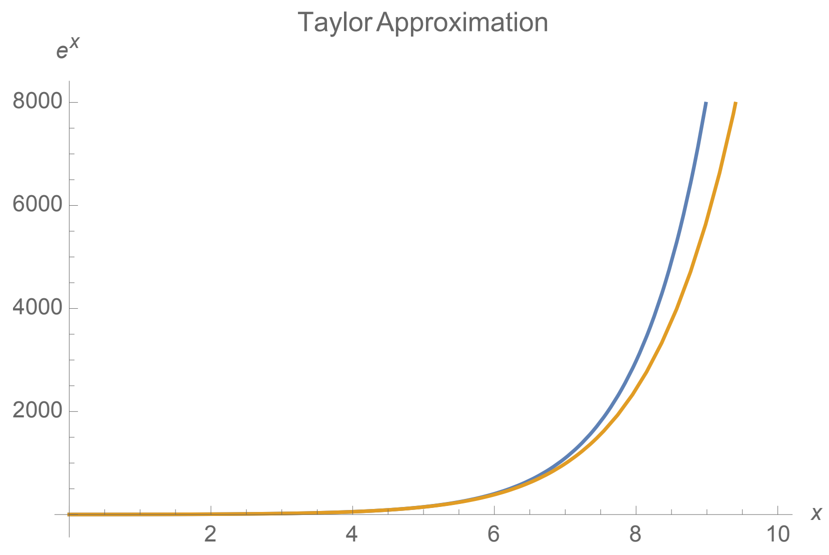

Figure 3: Comparing e^x with its 10-term Taylor approximation centered at zero.

If we are naïve about computing the terms of the series we can quickly get into trouble — the values of $k!$ get large *very quickly*. We can do better if we observe that:

$$\frac{x^k}{k!} = \frac{x^{k-1}}{(k-1)!} \times \frac{x}{k}.$$

At first, that looks like a recursive definition (and in fact, you could write it that way, but it would be wasteful). As we progress through the computation, assume that we know the previous result. We then just have to compute the next term and multiply it by the previous term. At each step we just need to compute $\frac{x}{k}$, starting with $k = 0! = 1$ and multiply it by the previous value and add it into the total. It turns into a simple `for` or `while` loop.

We can use an ϵ (epsilon) to halt the computation since $|x^k| \ll k!$ for a sufficiently large k. Consider Figure 4: x^k dominates briefly but is quickly overwhelmed by $k!$ and so the ratio rapidly approaches zero. The following is an approximation of e^x implemented in Python which halts computation at a default epsilon $\epsilon = 10^{-14}$. Make note of the efficient iterative computation of $x^k/k!$.

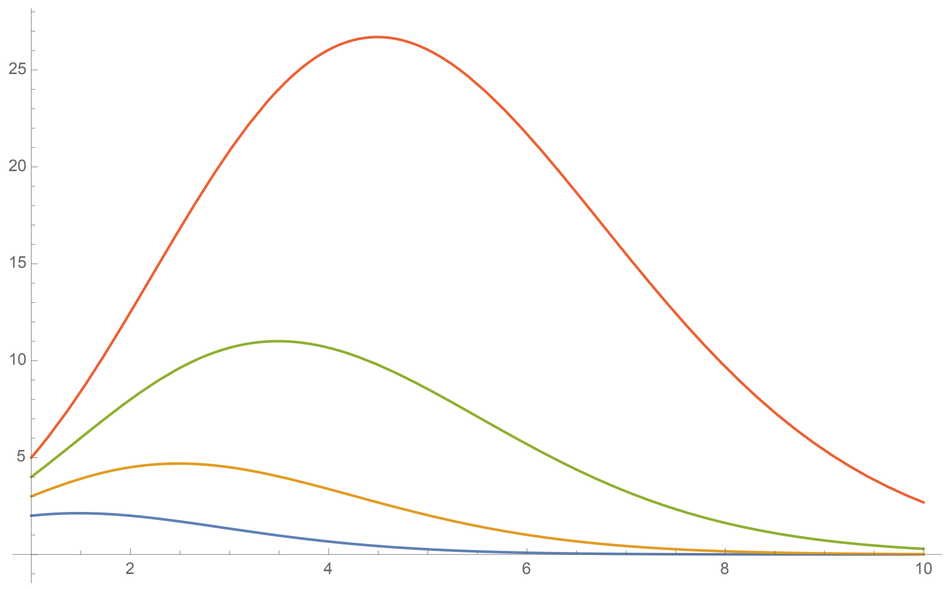

Figure 4: Comparing $\frac{x^k}{k!}$ for $x = 2, 3, 4, 5$.

Implementation of e^x

```python
def exp(x, epsilon = 1e-14):
    trm = 1.0
    sum = trm
    k = 1
    while trm > epsilon:
        trm *= abs(x) / k
        sum += trm
        k += 1
    return sum if x > 0 else 1 / sum
```

We take a different approach for $x < 0$ by noting that $e^{-x} = 1/e^x$. Why is that necessary? The series for $x \geq 0$ has all positive terms and so converges quickly, but for $x < 0$ it is an *alternating series* and converges much more slowly. Consequently, we do a little algebra and our computation is much more efficient.

5 Sine and Cosine

The sine and cosine functions repeat over the interval $[-2\pi, 2\pi]$. That is, $\sin(x) = \sin(x + 2k\pi)$ and $\cos(x) = \cos(x + 2k\pi)$ for every integer k. Centering their series around 0 makes computation simpler and more efficient.

The Taylor series for $\sin(x)$ centered about 0 is:

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

If we expand a few terms, then we get:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} + O(x^{14}).$$

We can implement this series as a simple loop. We will continue the loop until the last term is less than the error we have agreed is acceptable. Why not loop until it reaches *zero*? Think about it: `float` $\neq \mathbb{R}$.

Implementation of $\sin(x)$

```
def sin(x, epsilon = 1e-14):
    s, v, t, k = 1.0, x, x, 3.0
    while abs(t) > epsilon:
        t = t * (x * x) / ((k - 1) * k)
        s = -s
        v += s * t
        k += 2.0
    return v
```

The series for $\cos(x)$ centered about 0 is:

$$\cos(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}.$$

If we expand a few terms, then we get:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \frac{x^{12}}{12!} + O(x^{14}).$$

6 Square Roots and Logarithms

Can we just use the Taylor series to compute square root and the natural logarithm? Consider that $\sqrt{x} = x^{\frac{1}{2}}$, and so it is already a series but it has just *one term*. If we attempt a Taylor series we see that

$$\frac{d}{dx}\sqrt{x} = \frac{1}{2\sqrt{x}} \quad \text{and} \quad \frac{d^2}{dx^2}\sqrt{x} = -\frac{1}{4x^{3/2}} \quad \text{and} \quad \frac{d^3}{dx^3}\sqrt{x} = \frac{3}{8x^{5/2}} \quad \ldots$$

all of which contain $\sqrt{x}$, thus doing us no good at all. How about a series for $\log(x)$? Since $\log(0)$ is undefined, we will use $\log(x+1)$:

$$\log(x+1) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} + \frac{x^7}{7} + O\left(x^8\right).$$

It exists, but it converges *extremely slowly*.

To compute $\sqrt{x}$ and $\log(x)$, you will use Newton's method, also called the Newton-Raphson method. It is an iterative algorithm to approximate roots of real-valued functions: solving $f(x) = 0$. Each iteration $k+1$ of Newton's method produces successively better approximation:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

For example, consider computing $\sqrt{y}$ with Newton's method. That is, in order to solve for some $\sqrt{y}$, we are searching for a *non-negative* x such that $x^2 = y$. We can express this as finding the root of $f(x) = x^2 - y$, giving us:

$$x_{k+1} = x_k - \frac{x_k^2 - y}{2x_k} = \frac{y}{2x_k} + \frac{x_k}{2} = \frac{1}{2}\left(x_k + \frac{y}{x_k}\right).$$

Each guess x_{k+1} gives a successive improvement over the previous guess x_k. Your function `Sqrt()` should behave the same as `log()` from `<math.h>`: compute $\sqrt{x}$. The following is an example that implements Newton's method for computing square roots.

Implementation of $\sqrt{x}$

```
def sqrt(x, epsilon = 1e-14):
    z = 0.0
    y = 1.0
    while abs(y - z) > epsilon:
        z = y
        y = 0.5 * (z + x / z)
    return y
```

Your function `Log()` should behave the same as `log()` from `<math.h>`: compute $\log(x)$ $(\ln(x))$. The procedure is very much the same as it was for the square root example, the main difference being that $f(x) = y - e^x$, since e^x is the inverse of log, *i.e.* $\log(e^x) = x$. Another key difference is the value converges when $e^{x_i} - y$ is small, where x_0 is initially 1.0 and is used to compute better approximations. In order to implement this function, you will have to use your `Exp()` function.

Implementation of $\log(x)$

```
def log(x, epsilon = 1e-14):
    y = 1.0
    p = exp(y)
    while abs(p - x) > epsilon:
        y = y + x / p - 1
        p = exp(y)
    return y
```

6.1 Scaling

You can implement the $\log(x)$ and $\sqrt{x}$ functions directly, and if you do you will find that they work well for small x but fail when x increases. For $\sqrt{x}$ the algorithm is simply less efficient, but for $\log(x)$ it will fail for $x > 29$ due to floating-point numbers not being real numbers. What should we do? The answer is surprisingly simple: we scale the problem to a small interval.

In the case of $\sqrt{x}$ we will factor out all powers of *four*: let $x = 4^k \times a$. This is trivially true for $k = 0$, and you will see that it can be done for $k \geq 1$ if $x \geq 4$. Suppose $x = 48$, then $x = 4^2 \times 3$, with $a = 3$. Observe that $\sqrt{4^k a} = 2^k \sqrt{a}$. For every 4 that we factor out, we multiply by 2 after we have computed the $\sqrt{a}$. In our example, $\sqrt{48} = \sqrt{4^2 \times 2} = 2^2\sqrt{3} = 4\sqrt{3}$. Why 4? Four is a *perfect square*, so it is easier.

```
f = 1.0 # 2^0
while x > 1:
    x /= 4.0
    f *= 2.0 # 2^(k+1)
```

In the case of $\log(x)$ observe that $\log(x) = \log(a \times e^f) = \log(a) + \log(e^f) = f + \log(a)$ when $x = e^f \times a$. We factor out e, and each time add 1 to f until $a < e$. This allows us to search for $\log(a)$ over the small interval $[1, e)$. For example, to calculate $\log(917) \approx \log(e^7 \times 0.836195762413492) \approx 7 + \log(0.836195762413492) \approx 6.821107472256466$.

```
e = 2.7182818284590455 # Euler's constant
while x > e:
    x /= e
    f += 1.0
```

7 Your Task

Your first task for this assignment is to implement a small library of mathematical functions, mimicking those found in `<math.h>`. The interface for your math library is given in `mathlib.h`. The implementation of the library should go in `mathlib.c`. You are *strictly forbidden* to use any functions from `<math.h>` in your own math library. You are also forbidden to write a factorial function. Each of the functions you will write must halt computation using an $\epsilon = 10^{-14}$, which was discussed in §4.

The functions you are expected to implement are as follows:

`double Exp(double x)`

Returns the approximated value of e^x. Refer to §4 for specifics.

`double Sin(double x)`

Returns the approximated value of $\sin(x)$. Refer to §5 for specifics.

`double Cos(double x)`

Returns the approximated value of $\cos(x)$. Refer to §5 for specifics.

`double Sqrt(double x)`

Returns the approximated value of $\sqrt{x}$. Refer to §6 for specifics.

`double Log(double x)`

Returns the approximated value of $\log(x)$. Refer to §6 for specifics.

Your second task for this assignment is to write a dedicated program, `integrate`, that links with your implemented math library and computes numerical integrations of various functions using the *composite Simpson's 1/3 rule*.

`double integrate(double (*f)(double), double a, double b, uint32_t n)`

Computes the numerical integration of some function `f` over the interval [a, b]. This should be done with composite Simpson's 1/3 rule using `n` partitions. Note the special syntax for the parameter `f`. This syntax in **C** denotes that `f` is a pointer to a function – a *function pointer* – that takes a single `double` as its sole argument and returns a `double`. Using a function pointer allows us to use this `integrate()` function with *any* function that matches the function signature specified by `f`. Refer to §2.2.2 for the formula for the composite Simpson's 1/3 rule.

7.1 Testing

You will want to test each of your functions to make sure that they are producing correct values. In this case, you *can* and *should* compare the values that you compute with the ones from `<math.h>`.

You are strongly encouraged to do this by writing small test programs for each function. For some mysterious reason, students are reluctant to write such short programs and prefer to add `printf()` to their code, or sit in deep befuddlement when their final numbers are incorrect.

It is important to *independently verify* each important component. In the case of this program, those components are the replacements for functions from `<math.h>`. We include an example below.

Comparing our sine against the C library

```c
#include <math.h>
#include <stdio.h>

#define EPSILON 1e-14

static inline double Abs(double x) { return x < 0 ? -x : x; }

double Sin(double x) {
    double sgn = 1, val = x, trm = x;
    for (int k = 3; abs(trm) > epsilon; k += 2) {
        trm = trm * (x * x) / ((k - 1) * k);
        sgn = -sgn;
        val += sgn * trm;
    }
    return val;
}

int main(void) {
    for (double x = -2.0 * M_PI; x < 2 * M_PI; x += 0.1) {
        printf("sin(%3.2lf) = %+lf (%+20.19lf)\n", x, Sin(x), sin(x) - Sin(
    x));
    }
    return 0;
}
```

8 The Main Program

A few dud universes can really clutter up your basement.

—Neal Stephenson, *In the Beginning... Was the Command Line*

As stated in §7, you are expected to write a dedicated program, `integrate`, that computes the numerical integration of a function over a specified interval using the composite Simpson's rule. This program, implemented in `integrate.c`, must use `getopt()` and must accept the following command-line options:

- `-a` : Sets the function to integrate to $\sqrt{1-x^4}$.
- `-b` : Sets the function to integrate to $1/\log(x)$.
- `-c` : Sets the function to integrate to e^{-x^2}.
- `-d` : Sets the function to integrate to $\sin(x^2)$.
- `-e` : Sets the function to integrate to $\cos(x^2)$.
- `-f` : Sets the function to integrate to $\log(\log(x))$.
- `-g` : Sets the function to integrate to $\sin(x)/x$.
- `-h` : Sets the function to integrate to e^{-x}/x.
- `-i` : Sets the function to integrate to e^{e^x}.
- `-j` : Sets the function to integrate to $\sqrt{\sin^2(x)+\cos^2(x)}$.
- `-n partitions` : Sets the upper limit of partitions to use in the composite Simpson's rule to `partitions`. This should have a default value of 100.
- `-p low` : Sets the low end of the interval to integrate over to `low`. This *should not* have a default value and must be specified each time the program is run.
- `-q high` : Sets the high end of the interval to integrate over to `high`. This *should not* have a default value and must be specified each time the program is run.
- `-H` : Displays the program's usage and synopsis.

The functions you will integrate will be provided to you in the `functions.c` file in the resources repository. Each function is implemented using functions from *your* math library, so be sure to finish the library first.

You are given Table 1, as well as a working reference program in the resources repository, for you to compare the results of your numerical integrations against. Each function should be integrated using *even* numbered partitions from $2, 4, \ldots, n-2, n$, where n is the specified upper partition limit. The output of your program *must* be formatted as follow:

Table 1: Table of approximated values.

Integral	Low	High	Value
$\sqrt{1-x^4}$	0	1	0.87401918476405
$1/\log(x)$	2	3	1.118424814549702
e^{-x^2}	-10	10	1.772453850905508
$\sin(x^2)$	$-\pi$	π	1.545303425380133
$\cos(x^2)$	$-\pi$	π	1.131387027213366
$\log(\log(x))$	2	10	3.952914142858876
$\sin(x)/x$	-4π	4π	2.984322451168924
e^{-x}/x	1	10	0.2193797774265986
e^{e^x}	0	1	6.316563839027766
$\sqrt{\sin^2(x)+\cos^2(x)}$	0	π	3.141592653589797

```
$ ./integrate -a -p 0.0 -q 1.0 -n 10
sqrt(1 - x^4),0.000000,1.000000,10
2,0.812163891034571
4,0.852988388966857
6,0.862714108378597
8,0.866720323920874
10,0.868814915051592
```

The first line of output should be `<function>,<low>,<high>,<partitions>`. The subsequent lines should be `<partition>,<value>`, where `value` is the approximated integrated value. Commas should be used as the delimiting character. This format will help you plot your results in your assignment writeup.

The values that you compute should be close to Table 1. How close your approximations comes will depend on the number of intervals that you select. *Do not just choose a large number.* An important task in numerical analysis is to understand what is necessary to gain the desired accuracy. Indeed, if you make the number of intervals *too large* you will not only waste computing resources, but you may find that your accuracy *goes down.*

9 Deliverables

You will need to turn in the following source code and header files:

1. `functions.c`: This file is provided and contains the implementation of the functions that your main program should integrate.
2. `functions.h`: This file is provided and contains the function prototypes of the functions that your main program should integrate.
3. `integrate.c`: This contains the `integrate()` and the `main()` function to perform the integration specified by the command-line over the specified interval.
4. `mathlib.c`: This contains the implementation of each of your math library functions.

5. `mathlib.h`: This file is provided and contains the interface for your math library.

You may have other source and header files, but *do not make things over complicated.* Any additional source code and header files that you may use must not use global variables. You will also need to turn in the following:

1. `Makefile`:
   - `CC = clang` must be specified.
   - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified.
   - `make` must build the `integrate` executable, as should `make all` and `make integrate`.
   - `make clean` must remove all files that are compiler generated.
   - `make format` should format all your source code, including the header files.
2. `README.md`: This must use proper Markdown syntax. It must describe how to use your program and `Makefile`. It should also list and explain any command-line options that your program accepts. Any false positives reported by `scan-build` should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.
3. `DESIGN.pdf`: This document *must* be a proper PDF. This design document must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.
4. `WRITEUP.pdf`: This document *must* be a proper PDF. This writeup must include, at least, the following:
   - Graphs displaying the integrated values as you vary the number of partitions. You should be using `gnuplot` to produce these graphs. Attend section for examples of using `gnuplot` and other UNIX tools. An example script for using `gnuplot` to help plot your graphs will be supplied in the resources repository.
   - Analysis of the produced graphs and any lessons that you learned about floating-point numbers.

10 Submission

The illusion of self-awareness. Happy automatons, running on trivial programs. I'll bet you never guessed. From the inside, how can you?

—Vernor Vinge, *A Fire Upon the Deep*

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

11 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 3 §3.4 – 3.7
  - Chapter 4 §4.1 & 4.2 & 4.5
  - Chapter 5 §5.11 & 5.12
  - Chapter 7 §7.2
  - Appendix B §B4

In retrospect, come to think of it, isn't everything seen in retrospect? —Marty Feldman