Set 2: Numerical Integration

Aritra Biswas

1 Simpson's formula

We show that Simpson's formula for the integral $I = \int_a^b f(x)dx$ has a local error of $O(H^5)$ with $H = b - a$. We can express $f(x)$ as a Taylor series around $x = a$:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \frac{f^{(4)}(\eta)}{4!}(x-a)^4, \tag{1}$$

with $x, \eta \in [a, b]$. Integrating from a to b, we obtain:

$$I = f(a)H + f'(a)\frac{H^2}{2!} + f''(a)\frac{H^3}{3!} + f^{(3)}(a)\frac{H^4}{4!} + f^{(4)}(\eta)\frac{H^5}{5!}.$$

Using equation 1 to estimate $f(b)$ and $f(c)$, we can rearrange Simpson's formula:

$$\begin{aligned}
I_{simp} &= \frac{H}{6}f(a) + \frac{4H}{6}f(c) + \frac{H}{6}f(b) \\
&= \frac{H}{6}f(a) + \frac{4H}{6}\left(f(a) + f'(a)\frac{H}{2} + \frac{f''(a)}{2!}\left(\frac{H}{2}\right)^2 + \frac{f^{(3)}(a)}{3!}\left(\frac{H}{2}\right)^3 + \frac{f^{(4)}(\eta)}{4!}\left(\frac{H}{2}\right)^4\right) \\
&\quad + \frac{H}{6}\left(f(a) + f'(a)H + \frac{f''(a)}{2!}H^2 + \frac{f^{(3)}(a)}{3!}H^3 + \frac{f^{(4)}(\eta)}{4!}H^4\right) \\
&= \frac{H}{6}f(a) + \frac{4H}{6}\left(f(a) + f'(a)\frac{H}{2} + \frac{f''(a)}{2!}\frac{H^2}{4} + \frac{f^{(3)}(a)}{3!}\frac{H^3}{8} + \frac{f^{(4)}(\eta)}{4!}\frac{H^4}{16}\right) \\
&\quad + \frac{H}{6}\left(f(a) + f'(a)H + \frac{f''(a)}{2!}H^2 + \frac{f^{(3)}(a)}{3!}H^3 + \frac{f^{(4)}(\eta)}{4!}H^4\right) \\
&= f(a)H + f'(a)\left(\frac{4H^2}{6\cdot 2} + \frac{H^2}{6}\right) + f''(a)\left(\frac{4H^3}{6\cdot 2!\cdot 4} + \frac{H^3}{6\cdot 2!}\right) + f^{(3)}(a)\left(\frac{4H^4}{6\cdot 3!\cdot 8} + \frac{H^4}{6\cdot 3!}\right) \\
&\quad + f^{(4)}(\eta)\left(\frac{4H^5}{6\cdot 4!\cdot 16} + \frac{H^5}{6\cdot 4!}\right) \\
&= f(a)H + f'(a)\frac{H^2}{2!} + f''(a)\frac{H^3}{3!} + f^{(3)}(a)\frac{H^4}{4!} + f^{(4)}(\eta)\frac{5H^5}{576}.
\end{aligned}$$

This gives the error:

$$I - I_{simp} = -\frac{H^5}{2880}f^{(4)}(\eta) = O(H^5).$$

When we divide the interval $[a, b]$ into N subintervals of width $h_N = (b-a)/N$, we obtain the extended

Simpson's formula. Let $x_i = a + ih_N$, and let $m_{i,j} = \frac{x_i + x_j}{2}$.

$$\begin{aligned}\int_a^b f(x)dx &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \cdots \int_{x_{N-1}}^{x_N} f(x)dx \\ &\approx h_N \left(\frac{f(x_0) + 4f(m_{0,1}) + f(x_1)}{6}\right) + h_N \left(\frac{f(x_1) + 4f(m_{1,2}) + f(x_2)}{6}\right) + \cdots \\ &\quad + h_N \left(\frac{f(x_{N-1}) + 4f(m_{N-1,N}) + f(x_N)}{6}\right) \\ &= h_N \left(\frac{f(x_0) + 4f(m_{0,1}) + 2f(x_1) + 4f(m_{1,2}) + 2f(x_2) + \cdots + 2f(x_{N-1}) + 4f(m_{N-1,N}) + f(x_N)}{6}\right).\end{aligned}$$

The global error is the local error multiplied by the number of subintervals:

$$-\frac{h_N^5}{2880} f^{(4)}(\eta) \cdot N = -\frac{h_N^5}{2880} f^{(4)}(\eta) \cdot \frac{b-a}{h_N} = O(h_N^4).$$

2 Implementation of trapezoidal method

```
def trap(func, a, b, N):
    '''Integrates func(x) from a to b using the trapezoid method with N
    subintervals.'''

    if not is_python_or_numpy_int(N):
        raise TypeError('N is %s, must be an int or array of ints' % type(N))

    h = float(b - a) / float(N)

    # where to evaluate func
    ix = a + (h * np.arange(1, N)) # interior points
    ex = a + (h * np.array([0, N])) # exterior points

    # evaluate function at relevant points
    vfunc = np.vectorize(func) # version accepting array of x
    fval_ix = vfunc(ix)
    fval_ex = vfunc(ex)

    # sum function values
    sum_ix = np.sum(fval_ix)
    sum_ex = np.sum(fval_ex)

    # trapezoidal formula
    I = h * (sum_ix + (sum_ex / 2))

    return I
```

3 Implementation of Simpson's method

```
def simps(func, a, b, N):
    '''Integrates func(x) from a to b using the Simpson method with N
    subintervals.'''

    if not is_python_or_numpy_int(N):
        raise TypeError('N is %s, must be an int or array of ints' % type(N))

    h = float(b - a) / float(N)

    # where to evaluate func
    ix = a + (h * np.arange(1, N)) # interior regular points
    mx = a + (h / 2) + (h * np.arange(0, N)) # interior midpoints
    ex = a + (h * np.array([0, N])) # exterior points

    # evaluate function at relevant points
    vfunc = np.vectorize(func) # version accepting array of x
    fval_ix = vfunc(ix)
    fval_mx = vfunc(mx)
    fval_ex = vfunc(ex)

    # sum function values
    sum_ix = np.sum(fval_ix)
    sum_mx = np.sum(fval_mx)
    sum_ex = np.sum(fval_ex)

    # Simpson's formula
    I = (h / 6) * (sum_ex + 4*sum_mx + 2*sum_ix)

    return I
```

4 Comparison of trapezoidal and Simpson's methods

We use each of the above functions to evaluate $\int_0^1 e^x dx$ with N ranging from 10 to 10^3. Figure 1 shows the convergence plot for both methods on the same scale, which makes Simpson's method look as if it is independent of N relative to the trapezoidal method.

Clearly, Simpson's method is much more accurate at low N. Figure 2 shows the convergence plot for Simpson's method separately (but on the same x scale), showing that Simpson's method also converges faster (at $\sim 10^{1.6}$) than the trapezoidal method (which converges at $\sim 10^2$).

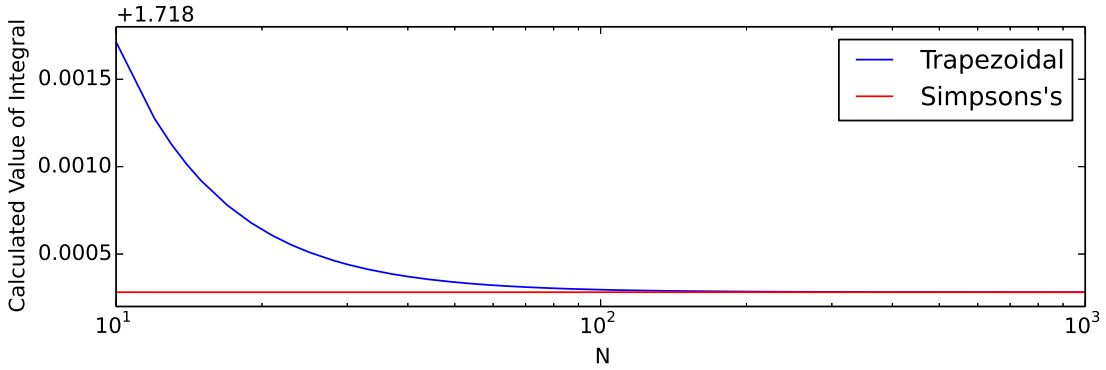

Figure 1: Convergence plot showing the effectiveness of the trapezoidal method and Simpson's method on the same scale.

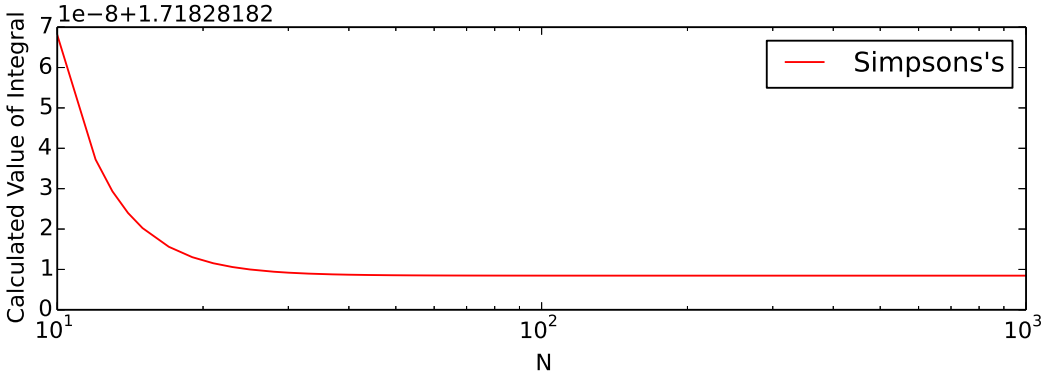

Figure 2: Convergence plot for Simpson's method on the same x scale as Figure 1.

5 Simpson's method to desired accuracy

We write a routine to integrate a function using Simpson's method repeatedly, starting with a predefined $N = N_0$, and doubling N each time until the relative difference between succesive approximations is less than the requested accuracy.

```
def simps_ac(func, a, b, ac):
    '''Integrates func(x) from a to b to the desired accuracy, ac, by
    repeating Simpson's method until successive iterations differ by less than
    ac.'''

    # have the initial number of subintervals scale roughly with the size
    # of the interval, but make sure it's at least a sensible size to begin
    # with
    N = 10 + int((b - a) / 5)

    # set initial guess
    I_old = 1

    while True:
        I_new = simps(func, a, b, N)
        if abs((I_new - I_old) / I_old) < ac:
            return I_new
        else:
            N *= 2
            I_old = I_new
```

We use this subroutine to calculate the integrals:

$$\int_0^1 e^x dx \text{ and } \int_1^2 x \sin\left(\frac{1}{x^2}\right) dx.$$

Since `simps_ac` simply runs the `simps` routine until the desired relative accuracy is obtained, the calculated value from `simps_ac` is the same as the value that `simps` would return for the corresponding value of N.

The routine checks relative accuracy, so higher values of `ac` will require many more iterations of Simpson's method. Since the desired accuracy is already provided, an interesting metric for this funtion's performance is the time required to calculate integrals to various values of `ac`. All times are total CPU times reported by IPython's `time` profiling.

Accuracy `ac`	Time for $\int_0^1 e^x dx$	Time for $\int_1^2 x \sin\left(\frac{1}{x^2}\right) dx$
10^{-5}	3.33 ms	3.33 ms
10^{-10}	6.67 ms	10 ms
10^{-13}	13.3 ms	26.7 ms
10^{-15}	36.7 ms	2.38 s

The routine can take a considerable amount of time to run at high accuracy, especialy when the function being integrated is rapidly changing, like $x \sin\left(\frac{1}{x^2}\right)$.

6 SciPy integration functions

As expected, `scipy.integrate.quad` and `scipy.integrate.romberg` are much faster in obtaining similar degrees of accuracy. We use these methods to calculate the two integrals in Section 5. In many cases, IPython reports a CPU time of 0 ns, i.e. too quick to measure accurately.

Measure	$\int_0^1 e^x dx$	$\int_1^2 x \sin\left(\frac{1}{x^2}\right) dx$
`quad` value	1.7182818284590453	0.6551059188460544
`quad` error bound	$1.9076760487502457 \times 10^{-14}$	$7.27313674671109 \times 10^{-15}$
`quad` time	0 ns	0 ns
`romberg` value	1.7182818284590782	0.65495874492202644
`romberg` time	0 ns	13.3 ms