MATH3511/6111: Scientific Computing
Mathematical Sciences Institute, Australian National University
Semester 1, 2022

Lab 3: Rounding Errors, Differentiation, Quadrature
Lindon Roberts
Due date: 9am, Monday 2 May (week 9)

This lab is based on earlier versions by Stephen Roberts, Graeme Chandler, Jimmy Thomson, Linda Stals and Kenneth Duru.

1 Representation of Numbers

1.1 Floating-Point Arithmetic

We know from lectures that floating-point numbers in binary are represented as

$$\pm m \times 2^e,$$

where $\pm$ is the sign, m is the mantissa and e is the exponent (an integer). Given a floating-point number, the function `math.frexp` calculates the corresponding values of m and e. Run this function for some example numbers and check that the answers are correct.

Warning! The convention in `math.frexp` is that $0.5 \leq m < 1$, which is different to our convention $1 \leq m < 2$ from lectures.

Since we store numbers to a fixed precision, there is a largest and smallest floating-point number (except for the special $\pm\infty$ constants in IEEE arithmetic), and a machine epsilon. NumPy provides lots of information about its built-in data types in `np.finfo`. For example, the machine epsilon of the default floating-point number can be found with

```
import numpy as np

# Get floating-point information for default float data type
float_info = np.finfo(float)
# Machine epsilon
print(float_info.eps)
```

Lab Book 1. Using `float_info` for the `float` data type (double precision), what is: (a) the largest positive finite machine number; (b) the largest (in absolute value) negative finite machine number; (c) the smallest positive machine number? You may need to read the documentation of `np.finfo` [3 points]

In IEEE arithmetic, there are some special constants for $\pm\infty$ and NaN (not a number). NumPy has these values as `np.inf` and `np.nan`.

```
# Generate np.nan
x = np.arange(5)
print(x / x)
# Generate np.inf
y = np.ones((5,))
print(y / x)
```

Warning! Be careful when you have `np.nan` results in 'if' statements:

```
# Is infinity equal to itself?
print(np.inf == np.inf)
# Is nan equal to itself?
print(np.nan == np.nan)
# Compare nan to other values
print(np.nan < 0, np.nan == 0, np.nan > 0)
```

By comparison, what happens if you calculate `0/0` or `1/0` in regular Python (not in NumPy)?

Write a function which calculates the Euclidean norm of a vector (i.e. $\|\boldsymbol{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$ if $\boldsymbol{x} \in \mathbb{R}^n$), but which excludes from the sum any entries which are NaN or $\pm\infty$. Test your function on: (a) a vector with no NaNs; (b) a vector with some NaNs; (c) a vector entirely consisting of NaNs.

1.2 Rounding Errors

When we do computing in floating-point arithmetic, we inevitably get rounding errors in our results. This is because any input numbers have infinite binary representations 'chopped' as soon as we input them. We can see some basic rounding errors with very simple calculations:

```
print(0.3 - 0.25 - 0.05)
print(0.2 + 0.4)
if 0.2 + 0.4 == 0.6:
    print("0.2 + 0.4 == 0.6")
else:
    print("0.2 + 0.4 != 0.6")
```

Generate your own examples of: (a) a simple floating-point calculation which gives (exactly!) the correct answer; (b) a simple floating-point calculation which does not give the correct answer (like the above).

One outcome of this is that checking `if x == y` is usually a bad idea for floating-point numbers. Instead, we can use `math.isclose` for numbers or `np.allclose` for vectors.

```
# Compare floating-point numbers
x = np.array([0.1+0.2, 0.2+0.4])
y = np.array([0.3, 0.6])
print(x == y)  # returns a vector of True/False, comparing x and y element-wise
print(np.allclose(x, y))  # use this to check if two vectors are (approximately) equal
```

If simple operations like addition can cause errors, it's not surprising that more complicated functions also have errors.

Lab Book 2. Find an example where `math.sin` or `math.cos` produces the wrong answer. How close is Python's calculation to the true value? Find an example where `math.sin` or `math.cos` produces exactly the right answer. [3 points]

If we have a calculation involving subtracting two similar numbers, catastrophic cancellation can occur and our answers can be very inaccurate. In lectures we saw this when trying to evaluate e^x for $x < 0$ using a Taylor series expansion. Here, we will try the function

$$f(x) = \frac{\cos(2x) - 1}{x^2}.$$

From a Taylor series expansion, we can show that

$$f(x) = -2 + \frac{2x^2}{3} + \mathcal{O}(x^4) \qquad \text{for } x \approx 0.$$

Lab Book 3. Evaluate $f(x)$ for $x = 10^{-2}, 10^{-3}, \ldots$ and compare your answer to the Taylor series. Comment on what you observe. Then find a better—but mathematically equivalent—way to evaluate $f(x)$. Check your new code is an improvement by comparing it to the Taylor series. [8 points]

Some mathematical expressions (which are susceptible to catastrophic cancellation) come up often enough that they have their own special implementations, such as `math.expm1` and `math.log1p`.

Lab Book 4. Look up the documentation of the function `math.expm1` and test it against the 'obvious' implementation using `math.exp` and a suitable Taylor series expansion. Comment on your results. [4 points]

2 Differentiation

Finite differencing gives us a way of approximating derivatives of a function. Recall the forward and central difference approximations:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \qquad \text{and} \qquad f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

Lab Book 5. By editing the code template below, use both approximations to estimate $f'(1)$ for the function $f(x) = e^x - \cos(x)$, for decreasing values of h. Estimate the order of convergence of both methods by plotting the approximation error (compared to the analytic derivative) against h. Comment on what you observe. [7 points]

```
# Decreasing sequence of h values
hs = 2**(-np.linspace(0,30,128))
# Make an empty vector of errors
forward_error = np.zeros((len(hs),))
central_error = np.zeros((len(hs),))

for i in range(len(hs)):
    h = hs[i]  # current value of h
    forward_error[i] = 0.0  # TODO edit this
    central_error[i] = 0.0  # TODO edit this

# The below vectors show the error decrease for particular orders of convergence
first_order_rate = hs      # plot hs vs first_order_rate to see O(h) convergence
second_order_rate = hs**2  # plot hs vs second_order_rate to see O(h^2) convergence rate

# TODO Plot hs vs forward_error and central_error (on a log-log plot)
```

Lab Book 6. Implement the second derivative estimation method from lectures to estimate $f''(1)$. Estimate the order of convergence by producing a plot similar to the above. [4 points]

2.1 Richardson Extrapolation

We can improve on the central difference method by using Richardson extrapolation, which combines the results of central differencing with different h values to produce an approximation which is better than each computed result. Using the code from the lectures, implement Richardson extrapolation to compute the derivative of the above function. Produce a table similar to slide 51 of the quadrature lectures, which compares forward differencing, central differencing and Richardson extrapolation.

2.2 Derivative testing

Another use of finite differencing is to check any code we have written to evaluate (exact) derivatives. Suppose we have written a simple function and also wrote code to evaluate its derivative (e.g. for use in Newton's method).

```
def f(x):
    return x**3 - 1.0 / x

def df(x):
    return 3.01*x**2 + 1.0 / x**2  # incorrect!
```

Then, we can compare finite differencing against our 'true solution' to check for an appropriate convergence rate:

```
x = 1
for i in range(8):
    h = 10**(-i)
    finite_diff = 0.0  # TODO implement forward differencing
    deriv_error = abs(finite_diff - df(x))  # compare to "true" derivative
    print("h = {0:^5.2e}, error = {1:^10.3e}".format(h, deriv_error))
```

Finish the above code and run it using both the above incorrect and the correct implementation of `df(x)`. Compare your results.

This is a very useful way to check for coding errors (either in `f(x)` or `df(x)`). Since we are often writing code to evaluate derivatives when we are doing rootfinding (or optimisation generally), SciPy has a function which can check your derivative code: `scipy.optimize.check_grad`.

3 Integration

The following code uses left Riemann sums to estimate $\int_a^b f(x)dx$. It is tested using an equally-spaced grid (with $h = (b-a)/n$), but the function works for non-equally spaced nodes too.

```
def riemann_sum(xdata, ydata):
    """
    Approximate an integral using Riemann sums, based on evaluated
    function values (x0,y0), ..., (xn, yn).

    Inputs xdata and ydata are (xi,yi), both vectors of length n+1
    """
    n = len(xdata) - 1
    if len(ydata) != n+1:
        print("Lengths of xdata and ydata do not match")
        return np.nan  # what do you think we should do here?
    # Calculate Riemann sum
    approx_integral = 0.0
    for i in range(n):
        approx_integral = approx_integral + (xdata[i+1] - xdata[i]) * ydata[i]
    return approx_integral

# Basic code to run riemann_sum(...)
# Assume we have a function f(x) using numpy element-wise functions,
```

```
# limits of integration [a,b], and number of points n
xs = np.linspace(a, b, n+1)  # equally spaced points xi
ys = f(xs)
print("Left Riemann sum =", riemann_sum(xs, ys))
```

Unfortunately, this code uses a 'for' loop over NumPy arrays. This can be very slow if n is large.

Lab Book 7. Rewrite `riemann_sum` using NumPy array operations, so that it does not have any loops, but still works for non-equally spaced nodes (you will need to use slicing from Lab 1). Use your new code to approximate $\int_{-1}^{1} e^{-x} dx$, and (with reference to theory from lectures) compare your approximation to the true value of the integral for increasing values of n and comment on your results. [5 points]

Lab Book 8. Using `riemann_sum` as a starting point, write a function which implements the composite trapezoidal rule which works for non-equally spaced nodes (without using 'for' loops). Compare the accuracy of the trapezoidal rule to left Riemann sums and comment on what you observe. [5 points]

3.1 Integration in SciPy

For a fixed collection of nodes, we can improve on the composite trapezoidal rule by using Newton-Cotes formulae. SciPy implements Simpson's Rule, the quadratic Newton-Cotes formula, in the function `scipy.integrate.simpson` (or `scipy.integrate.simps` in earlier SciPy versions). Use this function to calculate the same integral as above and compare to the methods you implemented yourself.

These methods are useful if we have already evaluated the integrand at specific points. However, we may not have data at the most useful points (e.g. we know equally spaced points is not good for polynomial approximation). The optimal choice of points comes from Gaussian quadrature. SciPy can integrate a function with Gaussian quadrature using `scipy.integrate.fixed_quad` — note that, unlike the `scipy.integrate.simpson`, you have to provide the integrand as an input, not a vector of values.

Lab Book 9. Use Simpson's rule and Gaussian quadrature (both with 20 points) to calculate the integral of Runge's function,

$$\int_{-1}^{1} \frac{1}{1+25x^2} dx = \frac{\arctan(5) - \arctan(-5)}{5} \approx 0.549,$$

and compare the results. [3 points]

SciPy also has more sophisticated integration options: `scipy.integrate.quad` is recommended as the default integration function, and `scipy.integrate.quadrature`, which don't use a fixed approximation degree. Instead, they keep increasing the approximation degree (evaluating the integrand at more and more points), until it estimates that the error is less than some tolerance (that you can choose) — this is called 'adaptive' quadrature. Both functions return two outputs: the approximate integral and an estimate of the error:

```
# Adaptive quadrature rules in SciPy
integral, error, infodict, message = scipy.integrate.quad(f, a, b)
integral, error = scipy.integrate.quadrature(f, a, b)
```

Lab Book 10. Use `scipy.integrate.quad` to integrate Runge's function as above for different absolute tolerance levels (set the relative tolerance to be very small). How good is the output error estimate to the true error? [3 points]

3.2 Monte Carlo Integration

A very different approach for integration is to use Monte Carlo methods, which are based on generating random numbers. NumPy has many different functions for generating random numbers. For example,

```
# Generate random values, each sampled uniformly from [0,1)
random_vector = np.random.rand(30)   # random vector of length 30
random_matrix = np.random.rand(5,5)  # random 5*5 matrix
# Generate random values, each from a standard normal distribution (mean 0, variance 1)
random_normal_vector = np.random.randn(30)   # random vector of length 30
random_normal_matrix = np.random.randn(5,5)  # random 5*5 matrix
```

Computers can't generate perfectly random numbers, they are generated using a 'pseudorandom' method. These are deterministic algorithms which produce values which are essentially indistinguishable from truly random numbers.

Sometimes, it is useful to force a program to generate the same set of random numbers multiple times (e.g. to test some code). You can do this by setting the 'random seed' (an integer which is used to start the pseudorandom algorithm). For example, this code produces two different random vectors:

```
random1 = np.random.rand(5)
random2 = np.random.rand(5)
print("First random vector =", random1)
print("Second random vector =", random2)
```

But this code prduces the same random vectors:

```
np.random.seed(38)
random1 = np.random.rand(5)
np.random.seed(38)
random2 = np.random.rand(5)
print("First random vector =", random1)
print("Second random vector =", random2)
```

The process for Monte Carlo integration is:

1. Generate N random points x_i, uniformly distributed over $[a, b]$
2. Evaluate each $f(x_i)$ and calculate the average value $\overline{f} = \frac{1}{N}\sum_{i=1}^{N} f(x_i)$.
3. The integral approximation is $(b-a)\overline{f}$.

This technique is useful for integrating in high dimensions, and is also widely used in mathematical finance (note that the expected value of a random variable is itself an integral).

Using Monte Carlo integration (and the `np.random.rand` function), estimate

$$\int_0^1 2x\sin(x) + x^2\cos(x)dx = \sin(1).$$

How does the error decrease as N increases? Try to write your code without using 'for' loops over NumPy arrays.

For a more complicated example, we can use Monte Carlo simulation to estimate the value of π:

1. Simulate N random 2d vectors with entries uniformly distributed in $[-1, 1]$.
2. The proportion of random vectors inside the unit circle ($x_1^2 + x_2^2 \leq 1$) is the ratio of the area of the unit circle to the area of the square.

This is equivalent to computing the Monte Carlo integral

$$\text{Area of unit circle} = \int_{-1}^{1}\int_{-1}^{1} f(x_1, x_2)dx_1dx_2, \qquad \text{where} \qquad f(x_1, x_2) = \begin{cases} 1, & x_1^2 + x_2^2 \leq 1, \\ 0, & \text{otherwise.} \end{cases}$$

Use this procedure to estimate the value of π, and study how the approximation error decreases as N increases.

Quasi-Monte Carlo Methods Recall from lectures that truly random (or pseudorandom) numbers tend to be 'clumped', and by using 'quasi-random' numbers which avoid clumping, we can improve the convergence rate of Monte Carlo methods. Quasi-random number generation has only recently (since version 1.7 in June 2021) been added to SciPy in the `scipy.stats.qmc` module.