Computational Lab 3 : Gaussian quadrature, numerical differentiation

Due September 27th, 2019 (@ 17:00)

- Read this document and do its suggested readings to help with the pre-labs and labs.
- The topics of this lab are to put into practice knowledge about numerical errors, and the trapezoid and Simpson's rules for integration.
- Ask questions if you don't understand something in this background material: maybe we can explain things better, or maybe there are typos.
- Whether or not you are asked to hand in pseudocode, you **need** to strategize and pseudocode **before** you start coding. Writing code should be your last step, not your first step.
- Test your code as you go, **not** when it is finished. The easiest way to test code is with `print('')` statements. Print out values that you set or calculate to make sure they are what you think they are.
- Practice modularity. It is the concept of breaking up your code into pieces that as independent as possible form each other. That way, if anything goes wrong, you can test each piece independently.
- One way to practice modularity is to define external functions for repetitive tasks. An external function is a piece of code that looks like this:

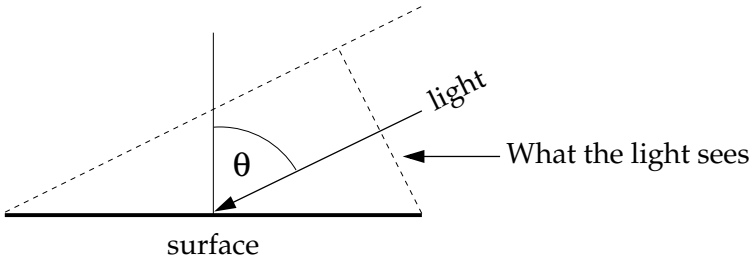
```
def MyFunc(argument):
    "''A header that  explains the function
    INPUT:
    argument [float] is the angle in rad
    OUTPUT:
    res [float] is twice the argument"''
    res = 2.*argument
    return res
```

  For example, in this lab, you will probably use Trapezoidal and Simpson's rules more than once. You may want to write generic functions for these rules (or use the piece of code, provided by the textbook online resources), place them in a separate file called e.g. `functions_lab02.py`, and call and use them in your answer files with:

```
import functions_lab02 as fl2  # make sure file is in same folder
ZehValyou = 4.
ZehDubble = fl2.MyFunc(ZehValyou)
```

Physics Background

Image processing and relief maps (adapted from Newman 5.23) When light strikes a surface, the amount falling per unit area depends not only on the intensity of the light, but also on the angle of incidence. If the light makes an angle θ to the normal, it only "sees" $\cos\theta$ of area per unit of actual area on the surface:



So the intensity of illumination is $a\cos\theta$, if a is the raw intensity of the light. This simple physical law is a central element of 3D computer graphics. It allows us to calculate how light falls on three-dimensional objects and hence how they will look when illuminated from various angles.

Suppose, for instance, that we are looking down on the Earth from above and we see mountains. We know the height of the mountains $w(x,y)$ as a function of position in the plane, so the equation for the Earth's surface is simply $z = w(x,y)$, or equivalently $w(x,y) - z = 0$, and the normal vector $\vec{v}$ to the surface is given by the gradient of $w(x,y) - z$ thus:

$$\vec{v} = \nabla[w(x,y) - \zeta] = \begin{pmatrix} \partial/\partial x \\ \partial/\partial y \\ \partial/\partial z \end{pmatrix} [w(x,y) - z] = \begin{pmatrix} \partial w/\partial x \\ \partial w/\partial y \\ -1 \end{pmatrix}. \tag{1}$$

Now suppose we have light coming in represented by a vector $\vec{a}$ with magnitude equal to the intensity of the light. Then the dot product of the vectors $\vec{a}$ and $\vec{v}$ is

$$\vec{a} \cdot \vec{v} = |\vec{a}||\vec{v}|\cos\theta, \tag{2}$$

where θ is the angle between the vectors. Thus the intensity of illumination of the surface of the mountains is

$$I = |\vec{a}|\cos\theta = \frac{\vec{a}\cdot\vec{v}}{|\vec{v}|} = \frac{a_x(\partial w/\partial x) + a_y(\partial w/\partial y) - a_z}{\sqrt{(\partial w/\partial x)^2 + (\partial w/\partial y)^2 + 1}}. \tag{3}$$

Let's take a simple case where the light is shining horizontally with unit intensity, along a line an angle ϕ counter-clockwise from the east-west axis, so that $\vec{a} = -(\cos\phi, \sin\phi, 0)$. Then our intensity of illumination simplifies to[1]

$$I = -\frac{\left[\cos\phi\,(\partial w/\partial x) + \sin\phi\,(\partial w/\partial y)\right]}{\sqrt{(\partial w/\partial x)^2 + (\partial w/\partial y)^2 + 1}}. \tag{4}$$

If we can calculate the derivatives of the height $w(x,y)$ and we know ϕ we can calculate the intensity at any point.

The relativistic particle on a spring: The energy of a relativistic particle, which is conserved, is given by

$$E = \frac{mc^2}{\sqrt{1 - \frac{v^2}{c^2}}} + \frac{1}{2}kx^2$$

which can be rearranged to show that

$$v^2 = c^2\left[1 - \left(\frac{mc^2}{(E - \frac{1}{2}kx^2)}\right)^2\right].$$

Assume the particle started from rest from an initial position which we will call x_0. In this case $E = mc^2 + \frac{1}{2}kx_0^2$ and after rearranging terms we can write the following expression for the positive root:

$$v = \frac{dx}{dt} = c\left\{\frac{\frac{1}{2}k(x_0^2 - x^2)\left[2mc^2 + \frac{1}{2}k(x_0^2 - x^2)\right]}{\left[mc^2 + \frac{1}{2}k(x_0^2 - x^2)\right]^2}\right\}^{1/2} = g(x), \tag{5}$$

[1] The sign convention for $\vec{a}$ and the next formula are opposite the sign convention used in the textbook. They ensure that light coming from the east $\phi \approx 0$ will result in positive intensity for a slope going upward to the west.

where $g(x)$ is a function of x. Notice that for $\frac{1}{2}k(x_0^2 - x^2) \ll mc^2$ we find $v \approx \sqrt{k(x_0^2 - x^2)}$ as expected for an energy conserving linear harmonic oscillator. For $\frac{1}{2}k(x_0^2 - x^2) \gg mc^2$, v approaches c but remains less than c.
Given (5), the period for the oscillation is given by four times the time taken for the particle to travel from $x = x_0$ to $x = 0$. Using separation of variables (see Example 5.10 in the text for a somewhat similar example):

$$T = 4\int_0^{x_0} \frac{dx'}{g(x')}. \tag{6}$$

In the small and large amplitude limits described above, we expect T to approach $2\pi\sqrt{k/m}$ and $4x_0/c$, respectively. In Q2 we will calculate T for a range of x_0 and compare it to these expected limits. Because $g(x) \to 0$ as $x \to x_0^-$, the integral diverges and a fairly large number of points will be required for an accurate calculation.

Computational Background

Solving Derivatives Numerically Section 5.10 deals with various methods for solving derivatives numerically. However, they assume you have a function which you can calculate at any point. If instead, you just have function values at different points, then the h' in the formulas has to be the distance between your data points. So for example, with a central difference scheme, to calculate derivatives you could use something like:

```
for i in range(xlen):
    dfdx[i] = (f[i+1]-f[i-1])/(2.0*deltax)
```

where `deltax` is the spacing between your points. However, notice that there will be an issue for the first and last `i` values. At `i=0`, `f[i-1]` doesn't exist. Similarly, at `i=xlen-1`, `f[i+1]` doesn't exist. So you can't use this formula for the endpoints. Instead, you need to use a forward difference scheme for the first point and a backward difference scheme for the last point.

Gaussian quadrature Section 5.6 of Newman provides a nice introduction to Gaussian quadrature. To summarize, Gaussian quadrature tweaks the Trapezoidal and Simpson Rule-type (i.e., Newton-Cotes) approach to integration in two important ways.

- First, instead of sampling a function at regularly spaced points, it finds optimal points to sample the function that will lead to a really good estimate of the integral.
- Second, instead of aiming for a specific order of errors, it describes an integration rule that is accurate to the highest possible order in a polynomial fit, which turns out to be a polynomial fit of order $2N-1$ for N sample points.

The cool thing is that if you happen to be integrating a polynomial function of less than order $2N-1$, Gaussian Quadrature with N sample points will be exactly correct (to within numerical roundoff). And it also works well with non-polynomial functions.
Example 5.2 on p.170 gives you the tools you need to get started. The files referenced are `gaussint.py` and `gaussxw.py`. To use this code, you need to make sure that both files are in the same directory (so the import will work). This code approximately calculates an integral according to the following formula:

$$\int_a^b f(u)\mathrm{d}u \approx \sum_{k=1}^{N} w_k f(u_k), \tag{7}$$

where a and b are limits of integration, and u is the (dummy) variable of integration. There are N so-called *sample points* $u_1, ..., u_N$ on the interval $a \le u \le b$. $f(u_k)$ is the function f measured at the sample point u_k. There are N coefficients called *weights*, $w_1, ..., w_N$. k is a dummy summation variable.
It is possible to write the previous integration formulas we've used before in this way. For example, for the Trapezoidal rule, the sample points are $u_1 = a$, $u_2 = a + h, \ldots$, $u_N = b$ and the weights are $w_1 = 1/2$, $w_2 = 1$, $w_3 = 1, \ldots$, $w_{N-1} = 1$, $w_N = 1/2$.
The line

```
x, w = gaussxw(N)
```

returns the `N` sample points `x[0]`, ..., `[N-1]` and the `N` weights `w[0]`, ..., `w[N-1]`. These weights and sample points can be used to calculate any integral on the interval $-1 < x < 1$. To translate this integral into one that approximates an integral on the interval $a < x < b$ you need to implement the change of variables formulas (5.61) and (5.62) of Newman, which are written in the code as

```
xp = 0.5*(b-a)*x + 0.5*(b+a)
wp = 0.5*(b-a)*w
```

The loop then sums things up into the summation variable `s`.
On p.171, there's a bit of code that lets you skip the change of variables by using `gaussxwab.py`, which you can use if you wish.
In Section 5.6.3 there's a discussion of errors in Gaussian quadrature, which are somewhat harder to quantify than for the previous methods we've seen. Equation (5.66) suggests that by doubling N we can get a pretty good error estimate:

$$\epsilon_N = I_{2N} - I_N. \tag{8}$$

Plotting lots of lines Here's a little trick to systematically display a lot of lines in a plot. Suppose you have a function of a dependent variable that depends on a couple of parameters. How can you plot a bunch of lines on the same plot to indicate a systematic dependence without having to laboriously construct plotting symbols? This is a great place to use python's `zip` functionality to pair plotting symbols or line types with parameter settings. E.g., try the following:

```
from numpy import pi, sin, arange
from pylab import plot, show, clf
clf()
phases = (0, pi/3)
colours = ('r', 'g')
amps = (1, 2, 3)
lines = ('.', '-', ':')
x = arange(0, 2*pi, 0.1)
for (phase, colour) in zip(phases, colours):
    for (amp, line) in zip(amps, lines):
       plot_str = colour + line
       plot(x, amp*sin(x+phase), plot_str)
show()
```

Factorial There is a function factorial in the math package that calculates the factorial of an integer. Your program might run out of memory if the numbers in $2^n n!$ get too large, so wrap them in floats as

```
float(2**n)*float(factorial(n))
```

Indications about the map question In that question, you will need to do some fiddling with plotting options to get a reasonable map. If you use the `imshow` routine, set `cmap='gray'` for the easiest view of the relief plot $I(x, y)$. You might need to show the transpose of the array depending on how you've read in your data. The `origin='upper'` or `origin='lower'` settings of `imshow` might also be helpful. If you want to make the coordinates clear for your plots, you can use the "extent" argument of imshow as in

```
imshow(..., extent=[west, east, south, north])
```

where the arguments indicate the west-east and south-north boundaries of the box.
In this map question, missing values are encoded in the file with large negative numbers, which suggests you need to use a narrow range for values with the `vmin` and `vmax` arguments to `imshow`. You might start with a narrow range for w and I and then adust the values until you get an informative image.

Solving Derivatives Numerically : Section 5.10 deals with various methods for solving derivatives numerically. However, they assume you have a function which you can calculate at any point. If instead, you just have function values at different points, then the 'h' in the formulas has to be the distance between your data points. So for example, with a central difference scheme, to calculate derivatives you could use something like:

```
for i in range(xlen):
     dfdx[i]=(f[i+1]-f[i-1])/(2.0*deltax)
```

where deltax is the spacing between your points. However, notice that there will be an issue for the first and last i values. At i=0, f[i-1] doesn't exist. Similarly, at i=xlen-1, f[i+1] doesn't exist. So you can't use this formula for the endpoints. Instead, you need to use a forward difference scheme for the first point and a backward difference scheme for the last point.

- forward difference:

  ```
  dfdx[i]=(f[i+1]-f[i])/deltax
  ```

- backward difference:

  ```
  dfdx[i]=(f[i]-f[i-1])/deltax
  ```

More on derivatives with central differences: The central difference approximation to the first derivative of $f = f(x)$ is

$$f'(x) \approx \frac{f(x+h/2) - f(x-h/2)}{h} \equiv \Delta_h f|_x,$$

where we have defined $\Delta_h f|_x$ as a central difference operator applied to f at x for a given interval h. Section 5.10 in the text points out that there is an optimum step size h for calculating derivatives using finite differences, including central differences. If h is too large the truncated Taylor series expansion becomes inaccurate. If h is too small the impact of numerical roundoff becomes large. So the optimum h balances numerical roundoff with algorithmic accuracy.
The central difference approximation to the second derivative of f is

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h/2)}{h^2} = \frac{\frac{f(x+h)-f(x)}{h} - \frac{f(x)-f(x-h)}{h}}{h} = \Delta_h(\Delta_h f|_x).$$

This means that repeated application of the central difference operator gives higher order derivatives: the mth derivative of f can be approximated as

$$f^{(m)}(x) \approx \Delta_h^m(f|_x),$$

where the superscript m indicates m applications of the operator on $f|_x$. For each higher order derivative, samples of f further from $x = 0$ are required. This effect, together with cancellation errors related to taking differences between numbers with similar values, serious affect the accuracy of this approximation.
The following code fragment uses recursion to implement this method:

```
# define function f(x) here
def my_f(x):
    return ... # return f given x


# calculate the m-th derivative of f at x using recursion.
def delta(f, x, m, h):
    if m > 1:
        return (delta(f, x + h/2, m - 1, h) - delta(f, x - h/2, m-1, h))/(h)
    else:
        return (f(x + h/2) - f(x - h/2))/(h)
```

```
# E.g. third derivative of my_f at x = 1.5 using h = 0.1.
# Notice how you can pass any function into this
d3fdx3 = delta(my_f, 1.5, 3, 0.1)
```

A very different approach to calculating higher order derivatives is given in the book in problem 5.22, using contour integration in the complex plane. The key result is that, from the Cauchy derivative formula, the mth derivative of a function $f(z)$ evaluated at $z = 0$ is given by

$$\left(\frac{d^m f}{dz^m}\right)_{z=0} \approx \frac{m!}{N} \sum_{k=0}^{N-1} f(z_k) e^{-i2\pi km/N},$$

where

$$z_k = e^{i2\pi k/N}.$$

This expression arises from the Trapezoidal rule and relates to the discrete Fourier transform, which can be solved efficiently using the fast Fourier transform method (FFT). We'll talk about the FFT later in the course.
We now have two methods to calculate higher order derivatives, and will compare them in Q2.

Questions

1. **[30%] Generating a relief map**

   The NASA Space Shuttle Radar Topography Mission (SRTM), which took place in 2000, recorded the elevation of the Earth's surface (with respect to the geoid) using synthetic aperture radar. Coarsened versions of the resulting data, which was combined with elevation data from other sources, are available at the US Geological Survey website

   http://dds.cr.usgs.gov/srtm/version2_1/SRTM3/

   as a list of files consisting of $1° \times 1°$ latitude-longitude tiles, each with 3 arcsecond resolution.[2] In each tile the elevation in height above sea level is packed in a binary format that you will need to unpack in this exercise. The format of the file names at the URL above is

   `<continent name>/<LAT><LON>.hgt.zip`

   where `<LAT>` is a latitude location like 50°N (with the name `N50`) and `<LON>` is a longitude location like 20°W (with the name `020W`). You will need to unzip the file to use it. The latitude and longitude locations mark the southwest corner of the tile. Each tile is a 1201×1201 grid of packed two-byte integers. The following code snippet would read the first couple of values from this file:

   ```
   import struct  # for reading binary files
   ...
   f = open(filename, 'rb')
   buf = f.read(2)  # read two bytes
   val1 = struct.unpack('>h', buf)[0]  # ">h" is a signed two-byte integer
   buf = f.read(2)  # read the next two bytes
   val2 = struct.unpack('>h', buf)[0]  # ">h" is a signed two-byte integer
   ```

   The order in which the file stores data is as follows: the first value `val1` is the elevation of the northwestern most point, the second value `val2` is the elevation of the point immediately to the east, and so on. The first 1201 values correspond to the northernmost latitude row from west to east, and the next 1201 values correspond to the latitude row immediately to the south of the first, and so on. You need to account for the north-to-south, west-to-east order of the data as you read it in.

[2] See http://dds.cr.usgs.gov/srtm/version2_1/SRTM30/srtm30_documentation.pdf

(a) Download the tile corresponding to the Island of Hawai'i (also known as the Big Island of Hawai'i). You'll need to explore the SRTM website a bit to find it (look in the "Islands" directory) . The file you download contains the elevation above sea level in metres.

Then write a pseudocode to do the following:

- Read and store the content of the input file into a two-dimensional array $w(x, y)$.
- Calculate the gradient of w, accounting for the different methods needed for interior and edge points. Explain your approach. In particular, devise a way to check that the borders are somewhat correctly treated (we do not seek a very careful method, a rough check will do).
- Create plots of w and I.

SUBMIT PSEUDO-CODE.

(b) Now implement this program. To calculate the derivatives you'll need to know the value of h, the distance in meters between grid points, which is about 420 m in this case. (It's actually not precisely constant because we are representing this part of the spherical Earth on a tat map, but $h = 420$ m will give reasonable results.) Create plots of both w and of I [from (3)]. For the I plot use $\phi = \pi$, corresponding to light shining from the west (kind of a sunset picture of the Earth's surface as scene from space). What is the main difference between the w and I maps? See if you can identify and zoom in on any well known features such as Mauna Loa, Mauna Kea, Kilauea, etc. Save and hand in any interesting zoomed images.

SUBMIT PLOTS, AND EXPLANATORY NOTES.

2. **[40%] The period of a relativistic particle on a spring**

Using Gaussian Quadrature, we will numerically calculate the period of the spring with the period given by (6), and see how it transitions from the classical to the relativistic case. The idea is to calculate T multiple times for a range of initial positions x_0 and assuming a mass of 1kg and a spring constant $k = 12N/m$.

(a) The first issue is accuracy of the solution. As x_0 gets smaller, the period should approach $2\pi\sqrt{m/k}$. Calculate the period for $N = 8$ and $N = 16$ for $x_0 = 1$ cm and compare this to the known classical solution. Estimate the fractional error for these two cases. To get a better sense of what affects the calculation, plot the integrand g_k and the weighted values $w_k g_k$ at the sampling points. Describe how these quantities behave as the x_0 limit of integration is approached. How do you think this behaviour might affect accuracy of the calculation?

HAND IN CODE, PLOTS AND WRITTEN ANSWERS.

(b) For a *classical* particle on a spring

$$m\ddot{x} = -kx,$$

what initial displacement $x_0 = x_c > 0$ for a particle initially at rest would lead to a speed equal to c at $x = 0$?

HAND IN YOUR WRITTEN ANSWERS.

(c) For $N = 200$, what is your estimate of the percentage error for the small amplitude case? Now plot T as a function of x_0 for x_0 in the range $1\text{ m} < x < 10x_c$, and compare it to the relativistic and classical limits as suggested at the beginning of the problem.

HAND IN PLOTS AND WRITTEN ANSWERS.

3. **[30%] Taking derivatives numerically**

(a) This part of the question deals with the impact of step size on the central difference calculation. Consider the function $f(x) = e^{2x}$, whose first derivative at $x = 0$ is $f'(0) = 2$. Using the central difference approximation, find this function's derivative numerically for a range of h's from $10^{-16} \to 10^{0}$ increasing by a factor of 10 each step (so you should have 17 h values with respective values of the derivative for each h value). Calculate the absolute value of the error in each derivative (i.e. the magnitude of the difference between your result and 2). Plot the error as a function of step size. Does the minimum in error correspond to the expected value from Section 5.10? Use $C = 10^{-16}$. If you want you can use the function `delta`, defined above, to do this calculation.

HAND IN YOUR CODE, ANY PLOTS, AND EXPLANATORY NOTES.

(b) This part of the question deals with higher-order derivatives. Write a program to calculate the first 10 derivatives of $f(x) = e^{2x}$ at $x = 0$ using the central difference function `delta` and the Cauchy derivative formula result as described above. As the book suggests for problem 5.22, use $N = 10000$ in the Cauchy formula. The mth derivative of f is 2^m so it will be easy to check your answers. The Cauchy derivative formula should give a result that consists of a complex number with a very small imaginary part; the real part should be the desired answer. Investigate the impact of step size on the central difference approach. Concisely describe your results and communicate what you think is happening. [*Hint: It doesn't really improve the central difference formula results to play with the step size, so we will not ask you to do a lot of work to investigate the impact of step size.*]

HAND IN YOUR CODE, ANY PLOTS, AND EXPLANATORY NOTES.

Question:	1	2	3	Total
Points:	30	40	30	100