Lab assignment #3: Gaussian quadrature, numerical differentiation

Nico Grisouard (Instructor), Alex Cabaj and Vince Pascuzzi (TAs),
Haruki Hirasawa (Marker)

Due Friday, 28 September 2018, 5 pm

- Read this document and do its suggested readings to help with the pre-labs and labs.
- 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.
- We are looking for "C^3" solutions that are **c**omplete, **c**lear, and **c**oncise. I.e., concisely and clearly explain the problem, the solution and numerical methods, and what is being shown in plots.
- 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
```

Once you wrote (or downloaded) the function(s), place them in a separate file called e.g. `MyFunctions.py`, and call and use them in your answer files with:

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

Computational Background

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:

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

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:

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

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))`.

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.

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.

Physics Background

Blowing snow: In atmospheric science, *blowing snow* is defined as snow that is lifted to a significant height above the surface, reducing visibility. An empirical formula for diagnosing the probability of blowing snow in the Canadian Prairies is[1]

$$P(u_{10}, T_a, t_h) = \frac{1}{\sqrt{2\pi}\delta} \int_0^{u_{10}} \exp\left[-\frac{(\bar{u}-u)^2}{2\delta^2}\right] \mathrm{d}u, \tag{3}$$

where u_{10} is the average hourly windspeed at a height of 10 m, the average hourly temperature is T_a in °C, and the snow surface age is t_h in hours. The mean wind speed is

$$\overline{u} = 11.2 + 0.365\,T_a + 0.00706\,T_a^2 + 0.9\ln(t_h) \tag{4}$$

[1] The following is from Chapter 3 of *Snow and Climate*, by Armstrong and Brun.

Figure 1: Soon, Toronto. Soon.
Image taken from https://en.wikipedia.org/wiki/Snowsquall

and the standard deviation of the wind speed is

$$\delta = 4.3 + 0.145 T_a + 0.00196 T_a^2. \tag{5}$$

The last two equations express how the mean wind speed tends to increase as the snow surface ages and becomes more smooth, and how some temperatures are more favourable to stronger and more variable winds than others.

Quantum uncertainty in the harmonic oscillator (problem extended from Newman 5.13)
In units where all the constants are 1, the wavefunction of the nth energy level of the one-dimensional quantum harmonic oscillator — i.e., a spinless point particle in a quadratic potential well — is given by

$$\psi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} e^{-x^2/2} H_n(x), \tag{6}$$

for $n = 0 \ldots \infty$, where $H_n(x)$ is the nth Hermite polynomial. Hermite polynomials satisfy a relation somewhat similar to that for the Fibonacci numbers,

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x). \tag{7}$$

The first two Hermite polynomials are $H_0(x) = 1$ and $H_1(x) = 2x$.

The derivative of the Hermite polynomials satisfy

$$\frac{dH_n(x)}{dx} = 2nH_{n-1}(x) \tag{8}$$

and as a result,

$$\frac{d\psi_n(x)}{dx} = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} e^{-x^2/2} \left[-xH_n(x) + 2nH_{n-1}(x)\right]. \tag{9}$$

The quantum uncertainty of a particle in the nth level of a quantum harmonic oscillator can be quantified by its root-mean-square position $\sqrt{\langle x^2 \rangle}$, where

$$\langle x^2 \rangle = \int_{-\infty}^{\infty} x^2 |\psi_n(x)|^2 \mathrm{d}x. \tag{10}$$

This is also a measure of twice the potential energy of the oscillator. A similar calculation tells us that in these units the momentum uncertainty is

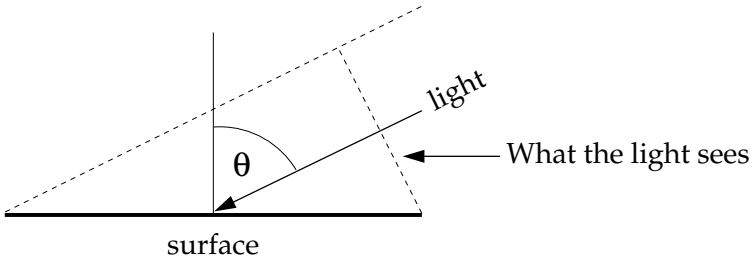
$$\langle p^2 \rangle = \int_{-\infty}^{\infty} \left| \frac{\mathrm{d}\psi_n(x)}{\mathrm{d}x} \right|^2 \mathrm{d}x, \tag{11}$$

which is a measure of twice the kinetic energy of the oscillator.

Summing potential and kinetic energy, the total energy of the oscillator is then

$$E = \frac{1}{2} \left(\langle x^2 \rangle + \langle p^2 \rangle \right). \tag{12}$$

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:

$$\vec{v} = \nabla[w(x,y) - z] = \begin{pmatrix} \partial/\partial x \\ \partial/\partial y \\ \partial/\partial z \end{pmatrix} [w(x,y) - z] = \begin{pmatrix} \partial w/\partial x \\ \partial w/\partial y \\ -1 \end{pmatrix}. \tag{13}$$

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

$$\vec{a}\cdot\vec{v} = |\vec{a}||\vec{v}|\cos\theta, \tag{14}$$

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

$$I = |\vec{a}|\cos\theta = \frac{\vec{a}\cdot\vec{v}}{|\vec{v}|} = \frac{a_x(\partial w/\partial x) + a_y(\partial w/\partial y) - a_z}{\sqrt{(\partial w/\partial x)^2 + (\partial w/\partial y)^2 + 1}}. \tag{15}$$

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[2]

$$I = -\frac{\left[\cos\phi\,(\partial w/\partial x) + \sin\phi\,(\partial w/\partial y)\right]}{\sqrt{(\partial w/\partial x)^2 + (\partial w/\partial y)^2 + 1}}. \tag{16}$$

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

Lab Questions

1. **([N%]) More on the error function**

   (a) In Lab 2, we evaluated the integral $(2/\sqrt{\pi})\int_0^3 \exp(-x^2)\mathrm{d}x$ using the Trapezoidal Rule and Simpson's Rule, as well as `scipy`'s method. We've now introduced Gaussian quadrature as a third method. Calculate the same integral with all three methods for a range of `N` slices/sample points between `N=8` and `N=1000`. *Note: you can just copy-and-paste a lot of what you did last week, or what I did in the solution (mention which one). Having all methods on the same document helps with clarity of the results.*

   SUBMIT PRINTED OUTPUT AND EXPLANATORY NOTES.

   (b) Calculate the relative error compared to the true value of erf(3) using these results and using (2), which requires you to do the calculation at `2N` as well as at `N`.

   Plot the magnitude of the relative error on a log-log scale as a function of `N`, and describe the results, e.g. the relative size of the errors, the validity of the error estimate (2), etc.

[2] 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.

SUBMIT FIGURE(S), AND EXPLANATORY NOTES.

(c) Re-cast eqn. (3) is terms of the error function erf, and use a Gaussian quadrature with `N=100` to calculate P from (3) for $u_{10} = (6, 8, 10)\,\mathrm{m\,s^{-1}}$ and t_h = (24, 48, 72) hours. Plot P as a function of T_a for these settings. You might want to use the plotting suggestions in the computational background to help come up with a compact way to display the information on a single line plot. How does the probability of blowing snow depend on the strength of the wind and the age of the snow? Does this dependence make sense to you? How does the temperature at which blowing snow is most likely to occur depend on the wind strength?

SUBMIT CODE, FIGURE(S) AND EXPLANATORY NOTES.

2. **([N%]) Calculating quantum mechanical observables**

(a) Write a user-defined function `H(n,x)` that calculates $H_n(x)$ for given x and any integer $n \geq 0$. Use your function to make a plot that shows the harmonic oscillator wavefunctions for n = 0, 1, 2, and 3, all on the same graph, in the range $x = -4$ to $x = 4$.

SUBMIT FIGURE AND EXPLANATORY NOTES (CODE TO BE SUBMITTED LATER).

(b) Make a separate plot of the wavefunction for $n = 30$ from $x = -10$ to $x = 10$.

Note: the program should take only a second or so to run.

SUBMIT FIGURE.

(c) Write a program that

- evaluates $\langle x^2 \rangle$, $\langle p^2 \rangle$ and energy E, as described in the Physics background, using Gaussian quadrature on 100 points, and then
- calculates the position and momentum uncertainty (i.e., the root-mean-square position and momentum of the particle) for a given value of n.

Use your program to calculate the uncertainty for $n = (0, 1, 2, \ldots, 15)$. What is the relationship between the uncertainty in position and the uncertainty in momentum? Do you notice a simple rule for the energy of the oscillator? *Note: you will need to use one of the evaluation techniques on p.179 of Newman to deal with the improper integrals here. For* $n = 5$, $\sqrt{\langle x^2 \rangle} \approx 2.35$.

SUBMIT CODE, PRINTED OUTPUT, AND EXPLANATORY NOTES.

3. **([N%]) 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.[3] 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)  # ">h" is a signed two-byte integer
buf = f.read(2)  # read the next two bytes
val2 = struct.unpack('>h', buf)  # ">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.

(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.

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

(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 flat map, but $h = 420$ m will give reasonable results.) Create plots of both w and of I [from (15)]. 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.