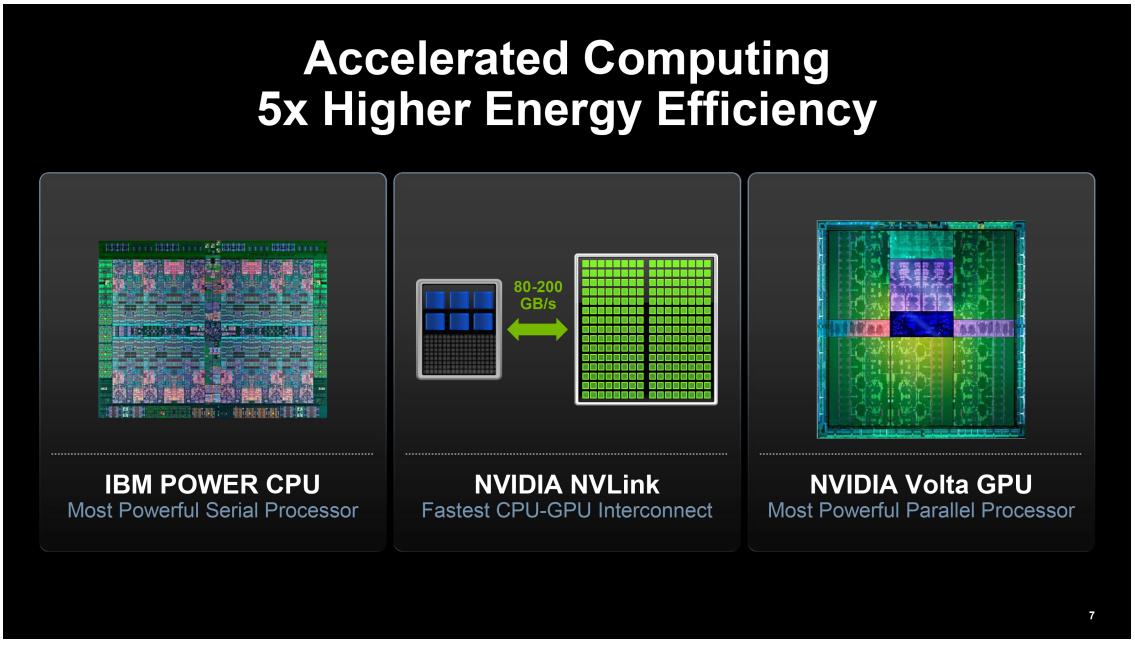
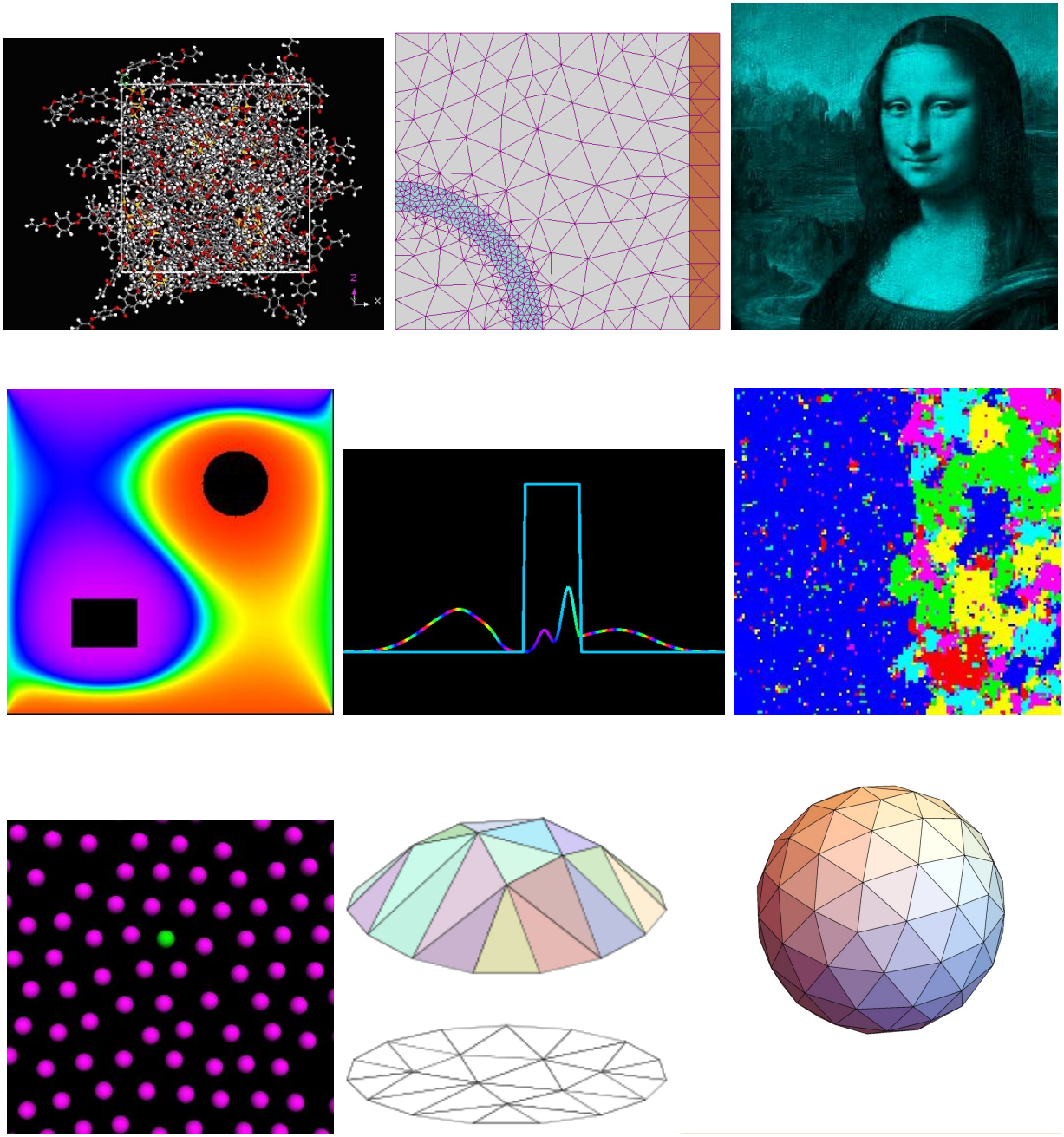
EC500 Paralllel Software for High Performance Computing

Rich Brower + Evan Weinberg

Course Description: The explosive advance in High Performance Computing (HPC) and advances in Big Data/Machine Learning and Cloud Computing now provides a fundamental tool in all scientific, engineering and industrial advances. Software is massively parallel so parallel algorithms and distributed data structures are required. Examples will be drawn from FFTs, Dense and Sparse Linear Algebra, Structured and unstructured grids. Techniques will be drawn from real applications to simple physical systems using Multigrid Solvers, Molecular Dynamics, Monte Carlo Sampling and Finite Elements with a final student project and team presentation to explore one example in more detail. Coding exercises will be in C++ in the UNIX environment with parallelization using MPI message passing, OpenMP threads and QUDA for GPUs. Rapid prototypes and graphics may use scripting in Python or Mathematica. (**Instructor: Prof. Richard Brower and Postdoctoral Fellow Evan Weinberg**)



Seven Dwarf: **Dense & Sparse linear algebra, Spectral methods, N-body methods, Structured & Unstructured grids and Monte Carlo methods**. These dwarfs have been identified as patterns, or motifs, which are the most important algorithm classes in numerical simulation



This is a "Hands on Course". Exercises will be started during class MW 12:20 - 2:05PM in the High Performance Computing Laboratory PHO 207. The "text" is on line lecture notes on Github and with access and documentation for advanced architecture: Programming guides for Intel Phi and Nvidia GPU. Prerequisite is programming experience in C at the level of EC327 or EC602 or consent of the instructor.

Contents

1 Lecture Notes: Introduction **1**

2 Lecture Notes: Derivative to Finite Differences **1**
2.1 Higher order and Undetermined Coefficients 2
2.2 Avoid Round off for $\Delta_h f(x)$ 2

3 Lecture Notes: Integrals to Sums **2**
3.1 Gaussian Integration 3

4 Lecture Notes: Root Finding **7**

5 Lecture Notes: Solving linear equation by Gaussian Elimination **1**

1 Lecture Notes: Introduction

These Lecture notes will be revised chapter by chapter to reflect the 2017 course and to respond to the interest and question of the students!

REVISION For Spring 2017: Lecture 1 & 2 Sun Jan 22 11:54:35 EST 2017

Preliminary Course Outline

1. First Third Week 1-4 (4 weeks)
   - Intro to HPC and Engineering impact
   - Numerical algebra for calculus
   - Differentiation & Integration (Gauss and Monte Carlo)
   - Newtons method for root finding & Non-linear optimization
   - Curve fitting &Error analysis
   - FFT as recursion for divide and conquer
   - Simple ODEs (oscillation vs relaxation)
2. Second Third Weeks 5-9 (inclusive 5 weeks)
   - Projects & Parallelization Method
   - MPI
   - OpenMP
   - CUDA and Data Parallel
   - Submitting to HPC systems.
   - Use of Libraries. ETC
3. Second Third Week 10-14 (5 weeks): Class Projects
   - Image Processing and Smoothness
   - CG and iterative solvers
   - MG solvers
   - MD short range (neighborhood tables) vs long range Coulomb
   - MonteCarlo for Magenets: Cluster and Graphs
   - FEM in 2D

References

There is no Text – Amazingly it hash't been written! The course materials and exercies will be posted on github. There are useful pieces in some books that I will try provide as a library in the Lab in PHO 207.

Grading

The grade is based on HW exercies (2/3) , a final project and participation in the class (1/3). The project will include code, performance analysis and write up and a presentation in the last week. The HW must be done individually but the project can be in a small team of 2 to 3 individuals.

Assignment 1: Floating Point Numbers
due: Jan 30, 2017 – 11:59 PM

GOAL: The main purpose of this exercise is to make sure everyone has access to the necessary computation and software tools for this class: a C compiler, a Python interpreter, the graphing program gnuplot, and for future use the symbolic and numeric evaluator Mathematica (though we won't be using that in this exercise). In this class, there will be help to make sure everyone's system is functional.

1 Background

1.1 Numerical Calculations

LANGUAGES & BUILT IN DATA FORMATS: The primary language at present used for high performance numerical calculations is C or C++. The standard environment is the Unix or Linux operating system. For this reason, C and Unix tools will be emphasized. We will introduce some common tools such as Makefiles and gnuplot.

That said, modern software practices take advantage of a (vast) variety of high level languages as well. We will use a bit of two interpreted/symbolic languages, Mathematica and Python, because of the power they have to develop, test, and visualize simple algorithms. Little prior knowledge except familiarity with C will be assumed.

In large scale computing all data must be *represented* somehow—for numerical computing, this is often as a *floating point* number. Floats don't cover every possible real number (there are a lot of them between negative and positive infinity, after all). They can't even represent the fraction 1/3 exactly. Nonetheless, they can express a vast expanse of numbers both in terms of precision as well as the orders of magnitude they span.

As you'll learn in this exercise, round off error, stability, and accuracy will always be issues you should be aware of. As a starting point, you should be aware of how floating points are represented. Each language has some standard built in data formats. Two common ones are 32 bit `floats` and 64 bit `doubles`, in C lingo. To get an idea of how these formats work on a bit-by-bit level, give a look at https://en.wikipedia.org/wiki/Single-precision_floating-point_format and https://en.wikipedia.org/wiki/Double-precision_floating-point_format.

You'll notice we're not shy about hopping off to the web to supplement the information we've put in this document and will discuss in class, and we expect you to do the same when you need to. **Searching the web is part of this course.**

As a last remark before we hop into some math, bear in mind that data types keep evolving. **Big Data** applications (deep learning!) are now using **smaller** 16 bit floats for a lot of applications. Why? Images often use 8 bit integer RGB formats. Some very demanding high precision science and engineering applications use 128 bit floats (quad precession). There are lots of tricks in code.

Symbolic codes represent some numbers like $\boldsymbol{\pi}$ and $\boldsymbol{e}$ as a special token since there are no finite bit representations! See for more about these issues: https://en.wikipedia.org/wiki/Floating_point.

1.2 Finite Differences

A common operation in calculus is the *derivative*: the local slope of a function. With pencil and paper, it's straightforward to evaluate derivative analytically for well known functions. For example:

$$\frac{d}{dx}\sin(x)\bigg|_{x=x_0} = \cos(x_0) \tag{1}$$

On a computer, however, it's a bit of a non-trivial exercise to perform the analytic derivative (you'd need a text parser, you'd need to encode implementations of many functions... there's a reason there are only a few very powerful analytic tools, such as Mathematica, that handle this). In numerical work the standard method is to approximate the limit,

$$\frac{df(x)}{dx} \equiv \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \tag{2}$$

with a finite but small enough difference h. The good news is this is completely general... the bad news is this is only an approximation and it is prone to errors. Due to round off, it's dangerous for h to get close to zero. $0/0$ is an ill-defined quantity!

One approximation of a derivative is the *forward finite difference*, which should look familiar:

$$D_h^+ f(x) = \frac{f(x+h)-f(x)}{h} \tag{3}$$

Two other methods are the backward difference and the central difference. The point of this exercise is to implement different types of differences, as well as test the effect of the step size h.

2 Programming Exercises

2.1 Part 1: Simple C++ Exercise

For this first exercise we've included the shell of a program below; it's your job to fill in the missing bits. The purpose of this program is to look at the forward, backward, and central difference of the function $\sin(x)$ at the point $x = 1$ as a function of the step size h. You should also print the exact dervative $\cos(x)$ at $x = 1$ in each column.

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;
```

```
double function(double x) {
  return sin(x);
}

double derivative(double x) {
  return cos(x);
}

double forward_diff(double x, double h) {
  return (function(x+h)-function(x))/h;
}

double backward_diff(double x, double h) {
  // return the backward difference.
}

double central_diff(double x, double h) {
  // return the central difference.
}

int main(int argc, char** argv)
{
  double h;
  const double x = 1.0;

  // Set the output precision to 15 decimal places (the default is 6)!

  // Loop over 17 values of h: 1 to 10^(-16).
  for (h = /*...*/; h /*...*/; h *= /*...*/)
  {
    // Print h, the forward, backward, and central difference of sin(x) at 1,
    // as well as the exact derivative cos(x) at 1.
    // Should you use spaces or tabs as delimiters? Does it matter?
    cout << /*...*/ << cos(1.0) << "\n";
  }
  return 0;
}
```

Don't be afraid to search online for any information you don't know! I'm not good at programming, I'm good at Googling and I'm good at debugging. You should name your C++ program `asgn1_findiff.cpp`. You can compile it with:

```
g++ -O2 asgn1_findiff.cpp -o asgn1_findiff
```

2.2 Part 2: Finite Differences, Python

Once you've written this program in C++, your next task is to rewrite it in Python! You should name your Python program `asgn1_findiff.py`. You should verify that the outputs agree exactly. If you save the output of each program to file (this is done with the character `>` on the bash command line), you can quickly verify the outputs agree with the bash program `diff`.

I'm currently learning Python myself from Codecademy. I'm developing my own scripts through a simple text editor, but for a more integrated development environment, you can look at Spypder: https://pythonhosted.org/spyder/. If you want an even more complete IDE, give a look at Anaconda: https://www.continuum.io/anaconda-overview. Stick with Python 3.5.

2.3 Part 3: Plotting using gnuplot

You have all of this data, now what? To visualize how the finite differences for different h compares with the analytic derivative, we can plot the data using the program `gnuplot`. Using the output file you generated with C or with Python (which should be equivalent!), plot the relative error in the forward, backward, and central difference as a function of h. This is similar to what is being plotted on the right hand side of Fig. 3 in the Lecture notes. As a reminder, the relative error is defined as:

$$\frac{|\text{approximate} - \text{exact}|}{|\text{exact}|} \tag{4}$$

Don't forget to set x and y labels on your graph, and titles for each curve in the key.

By default `gnuplot` will output to the screen. You'll want to submit an image at the end of the day; the commands `set terminal` and `set output` will be helpful in this regard! As an FYI: while it's best to play with making plots in the `gnuplot` terminal, it can get annoying to do everything there! `gnuplot` can just run a script file:

```
gnuplot -e "load \"[scriptname].gp\""
```

Where you should replace `[scriptname]` with, well, the name of your plotting script!

Extra Credit& Extra Fun: Once you have a program, it is good to see what else you can do. If you want to see a function that is difficult to numerically approximate, try the derivative of $\sin(1/x)$, which is exactly $-\frac{\cos(1/x)}{x^2}$, at some point close to zero, say $x = 0.0001$. Don't pass this in, but you can brag about in class for virtual extra credit.

2.4 Submitting Your Assignment

This first assignment is due at 11:59 pm on Monday, January 30. Please e-mail a **tarball** containing the assignment to the class e-mail, bualghpc@gmail.com. Include your name in the tarball filename.

If you're not familiar with tar, here's a sample instruction that perhaps I would use:

```
tar -cvf evan_weinberg_asgn1.tar asgn1_findiff.cpp asgn1_findiff.py asgn1_findiff.pdf
```

You may want to include other files (such as your gnuplot plotting script, though it's not required). **Do NOT include a compiled executable!** That's a dangerous, unsafe practice to get into.

A Software Tools: Nothing to Pass in!

A.0.1 Part I: Accessing BU Computing

Make sure you can access BU's "Linux Virtual Lab" by following the instructions here:

http://www.bu.edu/tech/services/support/desktop/computer-labs/unix/

These Linux machines aren't for running long calculations, but they are useful for small, interactive jobs. (I think any job will get killed after 15 minutes—don't quote me on that.) These machines also allow access to Mathematica.

Access to BU Computing is not a substitute for installing Mathematica and standard Unix compilers on your own machine, as described below.

A.0.2 Part II: Making sure you have a C++ compiler

In this class, we'll be using the standard compiler "g++". If you have a Mac or a Linux install, g++ may exist already. Try running the command:

```
which g++
```

from the terminal. On my machine, it returns:

```
/usr/bin/g++
```

But your mileage may vary. If it returns nothing, it means you don't have g++ installed, which you should go do! I'd be surprised if it wasn't installed, though.

If you're on Windows, you'll need to install Cygwin, which even I struggled with—other options are dual-booting, or a much better idea is installing Linux in a virtual box! If you're interested in that but not brave, send me (Evan) an e-mail at `weinbe2@bu.edu` and I'll help you out!

To make sure you understand compiling without an IDE (Integrated Development Environment), follow the quick tutorial here: https://en.wikibooks.org/wiki/C%2B%2B_Programming/Examples/Hello_world

Amazing Interactive Tutorias: C++

http://www.tutorialspoint.com/cplusplus/index.htm

A.0.3 Part Iii: Installing gnuplot

You may have gnuplot already installed on your machine. You can test this the same way we tested for g++:

```
which gnuplot
```

If it returns a path, you have gnuplot installed! If not, use your favorite package manager to install it. I'm an Ubuntu user, so I had to run:

```
sudo apt-get install gnuplot
```

If you're on a different distribution, you'll probably need to use `yum`, or some GUI tool. On Mac OS X, an optional package manager is Brew: http://brew.sh/, which will help you out.

By looking around on stackoverflow, I found a sample brew install command:

```
brew install gnuplot --wx --cairo --pdf --with-x --tutorial
```

Which will let you output PDFs as well as to the screen (that's the whole `with-x` and `wx`), I imagine. If you get stuck, let us know!

To test out gnuplot in OS X or Linux, run:

```
gnuplot
```

from the terminal. This will put you in an interactive gnuplot terminal. A few useful commands:

```
# Hashes aren't for twitter, they're for comments in gnuplot!
plot sin(x) # plot the sine function
f(x) = cos(x) # assign a function
plot sin(x), f(x) # plot two functions at once.
set xrange [0:2] # change the x axis.
set yrange [-2:2] # change the y axis range.
replot # update the plot with your new axis.
set yrange [-5:-2] # change the y axis range again.
replot # you won't see anything! So do...
reset # ... because you've messed up!
set xrange [-1:1]
plot x*sin(1/x) # This will lootk really bad!
set samples 1000 # sample the function more frequently.
replot # it should look a lot better now
exit # and we're done!
```

Your will want to save a figure from time to time. In this case before you exit add in these instructions.

```
set term postscript color  #one option that gives a .ps figure.
set output "myfigure.ps"   #whatever you want to name it
```

```
replot                          #send it to the output
set term x11                    #return to interactive view.
                                #On linux, you may need ‘‘wxt’’ instead of x11.
```

A.0.4 Part IV: Installing Mathematica

As students, you can luckily install Mathematica on your own computer without much pain. Follow this link and install Mathematica:

http://www.bu.edu/tech/support/desktop/distribution/mathsci/mathematica/student/step-1/

We've tested this on both Windows and Mac OS X. Mathematica will also work on standard Linux distros, we've just never tried installing it there ourselves—please try and let us know asap if you have issues.

After installing Mathematica, you should go through the following quick tutorials. They cover very simple topics, such as plotting, differentiation, and integration. The differentiation and integration articles go into much deeper mathematical detail than you'll need in this class! Just gleam out how to take a simple derivative and perform a simple integral. Don't let the word "Hessian" scare you.

- Plotting functions: https://reference.wolfram.com/language/tutorial/BasicPlotting.html
- Plotting data: https://reference.wolfram.com/language/howto/PlotData.html
- Differentiation: https://reference.wolfram.com/language/tutorial/Differentiation.html
- Integration: https://reference.wolfram.com/language/tutorial/Integration.html

We will suggest further reading as the need arises!

A.0.5 Part IV: Installing Anaconda for Python

You can get Anaconda distribution of python at

https://www.continuum.io/downloads

Get the one for Python 3.5 for your computer(s).

https://docs.python.org/2/tutorial/index.html

Amazing Interactive Tutorials:

https://docs.python.org/2/tutorial/index.html

http://www.tutorialspoint.com/python3/index.htm

http://docs.python-guide.org/en/latest/writing/style/#zen-of-python

2 Lecture Notes: Derivative to Finite Differences

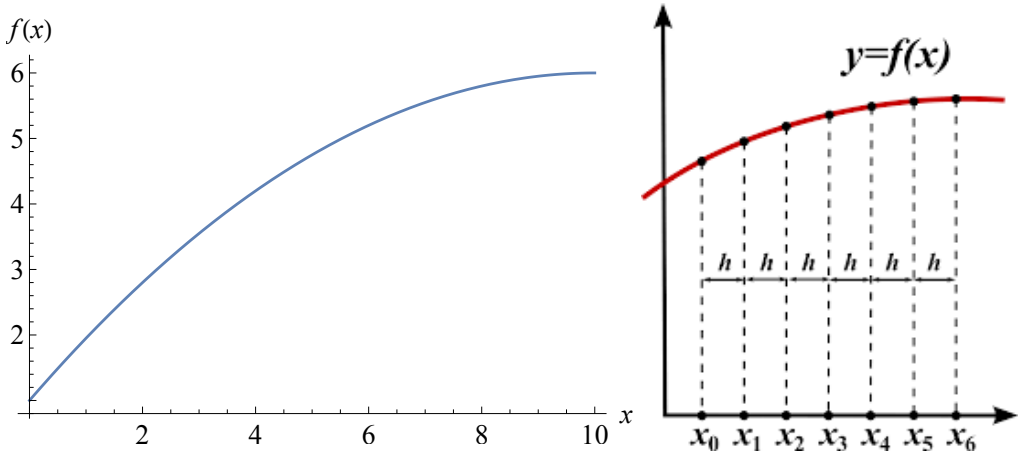


Figure 1: On left a function of x. On right the discrete points evalueted on a grid

We begin with the **1 D problems – The Derivative and the Anti-Derivative**: A smooth function $f(x)$ can be expanded in a power series around $x = 0$ expanded at x =0, (see Fig.1)

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots \ . \tag{1}$$

What is a_0? What is a_1? Hint:

$$\frac{dx^n}{dx} = nx^{n-1} \quad \text{or map} \quad Dx^n \to nx^{n-1} \tag{2}$$

So setting $x = 0$ we have $a_0 = f(0)$ taking the derivative and then setting $x = 0$ we get $df(0/dx = f'(0) = a_1$ and the next derivative $f''(0) = 2a_1$ etc. Ok now we have "derived" the Taylor series!

$$f(x) = f(0) + xf'(0) + \frac{1}{2!}x^2 f''(0) + \cdots + \frac{1}{n!}x^n f^{(n)}(0) + \cdots \ . \tag{3}$$

This is almost the only tool we will need from most of the course. In fact we will almost never use more than the first couple of terms.

Now we need to approximate a derivative on a computer. How? Define forward and backward difference approximation (see Fig.2)

$$\Delta_h f(x) = \frac{f(x+h) - f(x)}{h} \quad \text{and} \quad \widetilde{\Delta}_h f(x) = \frac{f(x) - f(x-h)}{h} \equiv \Delta_{-h} f(x) \tag{4}$$

NOTE: It is sometime convenient to take "units" where $h = 1$ (Much like just as in array notation in programming where x is a integer and the next element is $x + 1$). In this case I will woud write,

$$\Delta f(x) = f(x+1) - f(x) \quad \text{and} \quad \widetilde{\Delta} f(x) = f(x) - f(x-1) \tag{5}$$

If you are "smart" enough you can always figure out when and where to put back in the "lattice spacing" h.

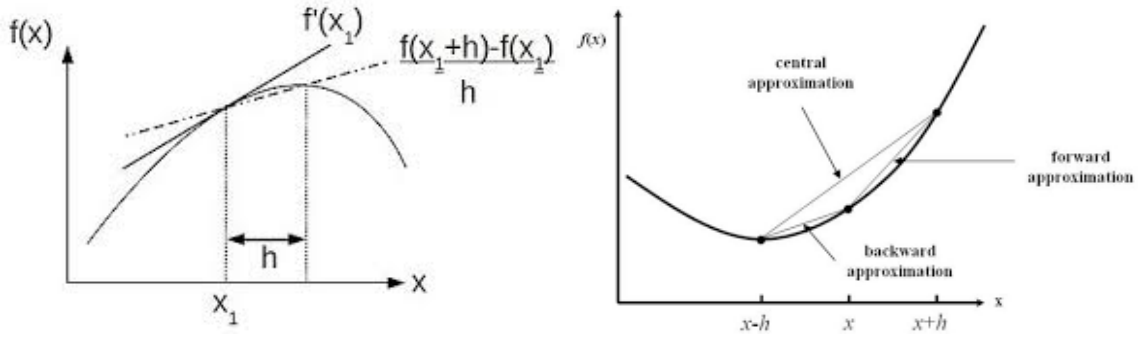


Figure 2: On left, finite difference. On right, central difference.

Note The second derivative is $\widetilde{\Delta}\Delta f(x) = \widetilde{\Delta}(f(x+1) - f(x)) = f(x+1) - 2f(x) + f(x-1) \simeq h^2 f''(x)$. Central difference is $\Delta + \widetilde{\Delta}$. As we will see later $\widetilde{\Delta} = -D^\dagger$.

2.1 Higher order and Undetermined Coefficients

Higher order approximations add $(\Delta_h + \Delta_h)/2h$ and $(\Delta_{2h} + \Delta_{2h})/4h$ to cancel $O(h^4)$ term.

2.2 Avoid Round off for $\Delta_h f(x)$

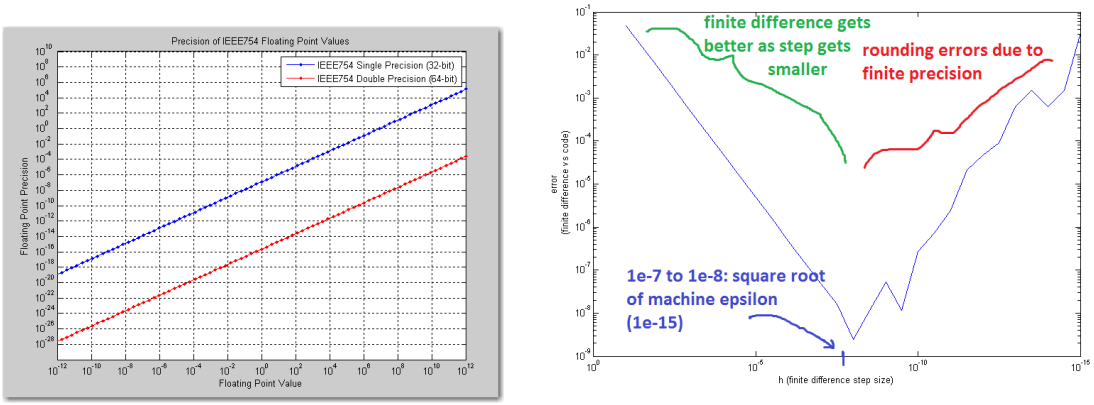


Figure 3: On the Right the error in the finite difference approximation to a deriative (y-axis) relative to size set size h (x-axis).

See IEEE floating point single (float) 32 bit is in this link:

https://en.wikipedia.org/wiki/Single-precision_floating-point_format

Consider the approximate derivative

$$df(x)/dx \simeq \Delta_h f(x) = \frac{f(x+h) - f(x)}{h} \tag{6}$$

It is very vulnerable to round off error. Let do this with no round off for x^N. (See NR Page 186 Sec 5.7) So to avoid round cancel the factor of h in the numerator and denominator explicitly,

$$\begin{aligned}\Delta_h x^N &= [x^N + Nhx^{N-1} + \cdots h^N - x^N]/h \\ &= Nx^{N-1} + \frac{N(N-1)}{2}hx^{N-1} + \cdots + h^{N-1}\,. \end{aligned} \tag{7}$$

The general expression for the coefficient in the expansion is the number of ways to put n things in i boxes: the coefficients in the expansion are

$$C[n][i] = \frac{n!}{(n-i)!i!} \quad \text{for} \quad i = 0, 1, \cdots, n \tag{8}$$

This is the way to get coefficients of: $(a+b)^0 = 1; (a+b)^1 = a+b; (a+b)^2 = a^2 + 2ab + b^2$, $(a+b)^3 = a^3 + 3a^2b + 3a^2b + b^3$ etc, which are just the values in the famous Pascal triangle:

$$C[n][i] = \begin{array}{c} n\backslash i= \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ \cdots \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & \cdots \end{array} \\ \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & \cdots \\ 1 & 1 & 0 & 0 & 0 & \cdots \\ 1 & 2 & 1 & 0 & 0 & \cdots \\ 1 & 3 & 3 & 1 & 0 & \cdots \\ 1 & 4 & 6 & 4 & 1 & \cdots \\ 1 & 5 & 10 & 10 & 5 & \cdots \\ \cdots & & & & & \end{array}\right] \end{array} \tag{9}$$

So we can avoid round off and compute (almost) exactly the finite difference form:

$$\Delta_h x^N = \sum_{i=1}^{N} C[N][i] x^{N-1} h^{i-1} = Nx^{N-1} + \frac{N(N-1)}{2!}x^{N-2}h + \cdots + h^{N-1} \tag{10}$$

computing coef $C[n][i]$'s recursively:

```
C[n][i] = 0;
for(n = 0; n<N+1; n++) C[n][0] = 1;
C[n+1][i+1] = C[n][i] +C[n][i+1];
```

Knuth's Concrete Mathematics Text has the same Pascal triangle format in Table 155 page 155 and the "addition formula" above in Eq 5.8 see Concrete Math(There is also a totally over the top number of random identities in this text. Fun but not that useful.)

Similarly we can define $\widetilde{\Delta}_h$ as an explicit expansion of $\widetilde{\Delta}_h$. What is? (Hint: Think what happen if $h \to -h$.) So the central difference cancels all term odd in h.

$$\frac{1}{2}(\Delta_h + \widetilde{\Delta}_h)x^N \tag{11}$$

This is good for any finite difference $\frac{1}{2}(\Delta_h + \widetilde{\Delta}_h)f(x)$ of course so the central difference has its first error in $O(h^2)$. See notes and code at https://github.com/BU-EC-HPC-S16/EC500-High-Performance-Computing

Assignment 2: Newton's Method and Root Finding
due: Feb 6, 2017 – 11:59 PM

GOAL: The purpose of Problem Set #2 is twofold: one purpose is to investigate searching and root-finding, but the other is to lay the groundwork for Problem Set #3: numerical integration, which has a lot to borrow from root finding, of all things!

The ultimate focus of this assignment looking for the zeroes of Legendre polynomials... this isn't made to scare you, but it's meant to emphasize how far you'll go in just a week. Finding the zeroes requires root finding, and if you're wondering what a Legendre polynomial is, well, it's a function we'll build recursively.

As a warm up to this impressive feat, we'll start of with simpler tasks: using *bisection* and *Newton's method* to find zeros of simple polynomials (which will apply to any continuous function, such as those pesky Legendre polynomials). And as a warm up to the warm up, we'll first point out that root finding is similar to searching in a sorted array!

Let's say you have a *sorted* array `c[0], c[1], ...`, and you want to find which index `i` satisfies `c[i] = b` for some `b`, that is, you want to find where `b` resides in the array. This is similar to finding the zero of a function—you're (discretely) solving:

```
c[i]-b = 0
```

With that in mind, let's get to it!

1 Warm Up Exercise : Searching an Array

To see the array version you should run the code `search.cpp` with the dependent `.h` files:

```
#include "search.h"
#include "sort.h"
```

that is on `github`. Read the code so you understand it. First set `Nsizse = 100` to see that it is all working. Then go back `= 10000000` to really see the differences between the different algorithms. This is a fun way to appreciate the advantages of better algorithms: LinearSearch, BinarySearch, and DictionarySearch are O(N), O(Log(N) and O(Log Log N) algorithms respectively. Or if you wish 3 algorithms: `Stupid, Smart, Brilliant!` Each has it corresponding version for root finding!

Modify the code `search.cpp` to run for a range of values of N (no larger than `10^8`) and output the results into a file `search_timing.dat`. The file should have four columns:

1. The value of N.
2. The number of iterations for linear search.
3. The number of iterations for binary search.
4. The number of iterations for dictionary search.

Using `gnuplot`, fit the curve to N, Log N and Log Log N to convince yourself that these estimates are right. (The demos on the gnuplot website might be helpful if you're not sure how to fit in `gnuplot`.) The deliverables for this exercise are:

- Your modified code, `search.cpp`.
- Your data file, `search_timing.dat`.
- Plots of your data with fit curves.

Do not stress too much about the plots of the data with fit curves. Don't worry about error bars and properly weighted fits! Later we will discuss proper weighted fits.

(Also note that MergeSort is a smart fast $O(NLog(N))$ sort. Try using a stupid $O(N^2)$ Selection Sort. It is so terrible you will have to run it over spring break or longer!)

2 Coding Exercise #1: Taking a Square Root

Again, we have a prewritten program for simple root finding. The code in `bisection_vs_newton.cpp` explicitly solves $f(x) = x^2 - A$, giving the positive square root of A. Just compile `bisection_vs_newton.cpp` and run. This program is a contest between bisection search, a `Log(N)` method, and Newton's method (think Dictionary search!), a `LogLog(N)` method. Try it for various values of A.[1]

[1] There are some weird cases where bisection will beat Newton's method—for example, the way the code is currently written, bisection will win for $A = 1024$. This is a special case due to how we set our initial guess and because 1024 is an even power of 2, namely 2^{10}. These cases are special!

This exercise has two pieces.

First, modify the code `bisection_vs_newton.cpp` to find the n-th root of A, instead of just the square root. This requires modifying the functions `bisection` and `newton` appropriately to take and use an additional numerical argument, `int n`, which you should prompt the user for. Once you are done, the code will find the zero of $f(x) = x^n - A$. For Newton, this requires the iteration

$$x \leftarrow x \times (1.0 - 1.0/n) + A/(n \times \text{pow}(x, n-1)). \tag{1}$$

Next, we'll focus specifically on square roots and third (cube) roots for $A = 2$. Instead of having the tolerance (`TOL`) fixed at the top of the code, make the tolerance a variable. Your goal is to study the iteration count as a function of the tolerance for `bisection` only. Sweep in N, where $N = 1/\texttt{TOL}$, for $N = 10, 100, \cdots 10^{15}$. Plot the iteration count as a function of N using gnuplot, and fit it to the form $c_0 + c_1 \log(N) + c_2 \log(\log(N))$, where c_0, c_1 and c_1 are free fit parameters.

The deliverables for this exercise are:

- Your modified code, `bisection_vs_newton.cpp`, which prompts the user for and computes arbitrary integer n-th roots using both `bisection` and `Newton's` method.
- Two plots showing the iteration count for computing the square root and cube root, respectively, of 2 using `bisection`. Your plots should include an overlaid fit curve.

For bit of extra credit and fun try to find the only zero of $f(x) = x^3(1 + \cos(10x)/2 - 1/4$ starting in the interval $[0, 3]$ starting with $x = 2$. Newton's method fails but bisection works. Why? (Are you having fun yet? https://www.englishforums.com/English/WhatMeantHaving/chll/post.htm)

3 Coding Exercise #2: Polynomials

3.1 Part #1: Generating Legendre Polynomials with Recursion

As a first step, we will generate the Legendre polynomials. Recall that an n-th order polynomial can be defined by its coefficients, $a_n[i] \equiv a[n][i]$ for $i = 0, \cdots, n$. To be more explicit, given these coefficients, the n-th order polynomial (such as the Legendre polynomial) can be evaluated as:

$$P_n(x) = a_n[0] + a_n[1]x + a_n[2]x^2 + \cdots + a_n[n]x^n \tag{2}$$

To specifically evaluate a Legendre Polynimial, all we need to do is construct the array $a[n][i]$. After a bit of searching on Wikipedia, we noted that the Legendre Polynomials satisfy a recurrence relation:

$$nP_n(x) = (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)\ . \tag{3}$$

The first two Legendre polynomials are defined as:

$$P_0(x) = 1 \qquad\qquad P_1(x) = x\ . \tag{4}$$

The recurrence relation and the definition of the first two Legendre Polynomials *uniquely* defines all P_n's. As an example, let's demonstrate finding the coefficients of $P_2(x)$. With complete generality, we can write:

$$P_2(x) = a_2[0] + a_2[1]x + a_2[2]x^2 \ . \tag{5}$$

We're looking to define the $a_2[i]$'s. We know the coefficients of the first two Legendre polynomials, so we can write:

$$P_0(x) = a_0[0] = 1, \tag{6}$$
$$P_1(x) = a_1[0] + a_1[1]x = 0 + (1)x, \tag{7}$$

where $a_0[0] = 1$, $a_1[0] = 0$, $a_1[1] = 1$. To find the coefficients of $P_2(x)$, we can plug these expressions into the recursion relation with $n = 2$:

$$\begin{aligned}
nP_n(x) &= (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x) \\
2P_2(x) &= 3xP_1(x) - 1P_0(x) \\
2\left(a_2[0] + a_2[1]x + a_2[2]x^2\right) &= 3x\left(a_1[0] + a_1[1]x\right) - 1\left(a_0[0]\right) \\
2\left(a_2[0] + a_2[1]x + a_2[2]x^2\right) &= 3x\left(x\right) - 1\left(1\right) \\
2\left(a_2[0] + a_2[1]x + a_2[2]x^2\right) &= 3x^2 - 1 \\
a_2[0] + a_2[1]x + a_2[2]x^2 &= \frac{3}{2}x^2 - \frac{1}{2}
\end{aligned}$$

We can now match powers of x on each side. This is known as *linear independence*: We thus get $a_2[0] = -\frac{1}{2}, a_2[1] = 0, a_2[2] = \frac{3}{2}$ and conclude:

$$P_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$$

We can generalize this, looking at each power x^i in Eq. 3, and find the recursion relation

$$n\ a[n][i] = (2n-1)\ a[n-1][i-1] - (n-1)\ a[n-2][i] \tag{8}$$

where $a[n-1][-1] = 0$. Why? Your task in this programming exercise is to write a function in `C` that finds the coefficients $a_n[0], a_n[1], \cdots, a_n[n]$ for a general n. If you carefully follow the steps I've given for $n = 2$, you'll see I already wrote it for you! Of course, you should assume that $a_n[i] = 0$ if $i > n$. You may find it useful to represent this in `C` as a two-dimensional array.

The function should be declared as:

```
int getLegendreCoeff(double* a, int n);
```

where:

- `a` is an array of `double`s, already allocated, of length `n`+1.
- `n` indicates the order of Legendre polynomial.
- The return value is 1 if there are no errors, 0 if there are errors. Some errors include:

- `n`< 0, since we have only defined the Legendre polynomials of integer order. (Yes, there are non-integer extensions. Check them out in Mathematica!)
- `a` is a null pointer.

This function should be included in a file `legendre.c`, with a main function that prompts the user for a value n, allocates an array, finds the coefficients of $P_n(x)$, and prints the polynomial. As an example, here's how the code may run, where $>$ indicates lines with user input. Assume we've started from a bash shell.

```
> ./legendre
What order Legendre polynomial?
> 5
7.875 x^5 + 0 x^4 + -8.75 x^3 + 0 x^2 + 1.875 x^1 + 0 x^0
```

We're not stressed about how many decimal points your output exhibits (as long as it's at least 6 if there are more than 6 non-zero digits).

The deliverables for this exercise are:

- Your own code file `legendre.c` as defined above with the function `getLegendreCoeff`.

3.2 Part #2: Finding Zeros of Legendre Polynomials

As a next step, we need to find the zeros of the Legendre polynomial. There are many different ways to do this, but for this assignment we will focus on the Newton-Rhapson's method. In general, the Newton-Rhapson method does not always converge to a zero of the function—there are some pathalogical cases where the method fails. We won't go very in depth on this; for more details, check out:

- https://en.wikipedia.org/wiki/Newton%27s_method, namely the section "Failure of the method to converge to the root".

Luckily, the roots of the Legendre polynomial are "well-behaved," in large fact because they are orthogonal polynomials. They are so well-behaved that there are high-quality approximate expressions for the roots of general Legendre polynomials that serve as good initial guesses to Newton's method.

We will state without proof that the kth root of $P_n(x)$ is approximated by the expression:

$$\xi[n][k] \simeq \left(1 - \frac{1}{8n^2} + \frac{1}{8n^3}\right) \cos\left(\pi \frac{4k-1}{4n+2}\right). \tag{9}$$

Since the roots are almost equally spaced in $[-1, 1]$, these approximate roots lead to a quick convergence. For this assignment, write a routine:

```
int getLegendreZero(double* zero, double* a, int n);
```

where:

- `zero` is a pre-allocated array of length n where the (sorted!) zeroes will go when the function returns.
- `a` is a pre-allocated array of length $n+1$ which contains the coefficients $a_n[i]$ (which you'd compute from `int getLegendreCoeff(double* a, int n)`, I imagine)!
- `n` is, well, n.

As noted, use the "smart" initial guesses given above to populate the array `zero` with the zeroes of the given Legendre polynomial. You might want to test your code for $n = 5, n = 6$, and $n = 7$. While there are an infinite number of Legendre polynomials, your computer (as well as mine) doesn't have an infinite amount of memory, so you can have your code print an error if n is bigger than, say, 30, and of course print an error if n is negative.

Here's a sample output:

```
> ./getZeros
What order Legendre polynomial?
> 5
-0.9061798459386640 -0.5384693101056831 0.0000000000000000
0.5384693101056831 0.9061798459386640
> ./getZeros
What order Legendre polynomial?
> 31
C'mon man, don't ask me for an order greater than 30.
```

You can verify the program to the known values for the zeroes, listed (for example) here. In Problem Set # 3, we'll reuse these functions to construct arbitrarily accurate numerical integration routines using Gaussian quadrature.

The deliverables for this exercise are:

- Your own code file `getZeros.c` as defined above with the function `getLegendreZero`.

3.3 Submitting Your Assignment

This assignment is due at 11:59 pm on Monday, February 6. Please e-mail a **tarball** containing the assignment to the class e-mail, bualghpc@gmail.com. Include your name in the tarball filename.

If you're not familiar with tar, here's a sample instruction that perhaps I would use:

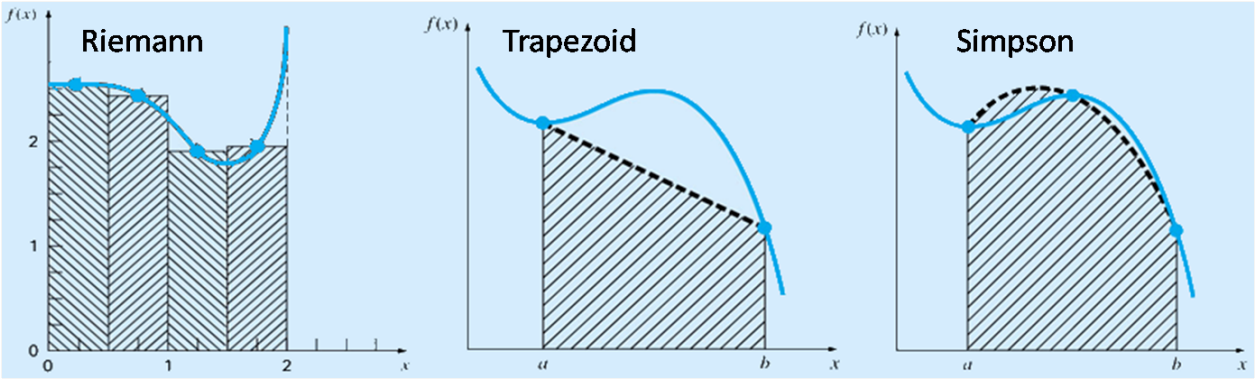
```
tar -cvf evan_weinberg_asgn2.tar [directory with files]
```

You may want to include other files. **Do NOT include a compiled executable!** That's a dangerous, unsafe practice to get into.

x

3 Lecture Notes: Integrals to Sums

In **Numerical Recipes in C** at http://apps.nrbook.com/c/index.html there is a nice introduction to numerical integration: the trapazoidal rule, Simpson's rule and Gaussian Integration.See Ch 4 Sec 4.1 - 4.2 and these lecture on Gaussian integration.



The solution(s) to the equation

$$Dy(x) = \frac{dy}{dx} = f(x) \quad \Longrightarrow \quad y(x) = \int_0^x f(x')dx' + c\,. \tag{12}$$

The derivative maps any function $g(x) \to (Dg)(x) = g'(x) = f(x)$. However since addding a constant gives the same derivative it is "many to one". The integral is really the inverse but with an unknow constant. We define D^{-1} to be the integral up to an unknown constant. We can call this an "inverse Derivative"

$$D^{-1}f(x) = g(x) = \int_0^x f(x')dx' + c \tag{13}$$

Check it (aka prove it!) : $D(D^{-1})f(x) = D\int_0^x f(x')dx' = f(x)$.

Now lets do this for finite steps. For sums we can solve the analagous expression:

$$\Delta_h y(x) = f(x) \quad \Longrightarrow \quad y(Nh) = \sum_{i=0}^{N-1} f(ih)h + c; \tag{14}$$

Why ? Ok how do we integrate in general? Setting $h = 1$ the expression our guess for Δ^{-1} is

$$\Delta^{-1}f(x) = f(x-1) + f(x-2) + \cdots + f(0) + c \tag{15}$$

Is this right? Let's check it.

$$\begin{aligned} \Delta\Delta^{-1}f(x) &= \Delta[f(x-1) + f(x-2) + \cdots + f(0) + f(-1) + c] \\ &= f(x) - f(x-1) + f(x-1) - f(x-2) + \cdots f(1) - f(0) + f(0) - f(-1) \\ &= f(x) \end{aligned} \tag{16}$$

All terms cancel except the first, if we take $f(-1) = 0$ as a convention and so we we could add it to "definition" of Δ^{-1}. *After all it gives no contribution to the integral since is not in the interval. This is in fact what we assume below doing the trapezoidal rule.*

Again as in the finite difference discussion, $\Delta \equiv \Delta_h$ with $h = 1$. This is often conveniet to avoid clutering up the formulae. It is simple to put h back in when it needed.) For this one time I'll put the h's

back in:

$$\begin{aligned}\Delta_h \Delta_h^{-1} f(x) &= \Delta[f(x-h)+f(x-2h)+\cdots+f(0)+f(-h)+c]h \\ &= (f(x)-f(x-h))+(f(x-h)-f(x-2h)+\cdots f(1)-f(0)+f(0)-f(-h) \\ &= f(x) \end{aligned} \tag{17}$$

Now let's be more systematic about looking for a good approximation to the integral. To approximate the integral

$$I=\int_b^a f(x)dx \to I_N=\sum_{i=1}^{N} w_i f(x_i) \tag{18}$$

We can pick good points x_i in the interval $[a,b]$ and nice weights.

The first method is call the Trapezoidal rule. Let's try it for a regular spacing $\Delta x \equiv h = (b-a)/N$ or $x_i = a + i\Delta$ for $(x_0, x_1, ..., x_N)$

$$\begin{aligned} y &= \sum_{i=0}^{N-1} h[f(a+ih)+f(a+(i+1)h)]/2 \\ &= h[\frac{1}{2}f(x_0)+f(x_1)+f(x_2)\cdots f(a+x_{N-1})+\frac{1}{2}f(x_N)] \end{aligned} \tag{19}$$

(see NR page 131, 4.1). Or in general Newton-Cotes for trapezoidal rule:

$$\int_{x_1}^{x_2} f(x) = h[f_1+f_2]/2 + O(h^3 f'') \tag{20}$$

Exact for any $f(x) = c_0 + c_1 x$ of course. With two free "weights," we can make any two terms exact: $\int_{-h}^{h} f(x) = h[c_{-1}f(-h) + c_1 f(h)]$ so $f = 1$ implies $2h = 2h[c_{-1} + c_1]$, and $f = x$ implies $c_{-1} = c_1$ so $c_{-1} = c_1 = 1/2$. Note x^{odd} gives zero automatically if we choose the intervale and coeffiicients symmtrically around $x = 0$. This implies

$$y_N = \sum_{i=0}^{N-1} (x_{i+1} - x_i)\frac{f(x_i)+f(x_{i+1})}{2} \tag{21}$$

Note that we don't need to assume that all the intervals are equal: $(x_{i+1} - x_i) \neq h$. This only exact for a straight line (linear) function $(f(x) = c_0 + c_1 x)$ There are better "higher order" fits. A next example is"Simpson's rule," which uses another interior point and is good up to $O(x^3)$ exactly:

$$\int_{-h}^{h} f(x)dx = h\left[\frac{1}{3}f(-h)+\frac{4}{3}f(0)+\frac{1}{3}f(h)\right]+O(h^5 f^{(4)}) \tag{22}$$

Proof. Odd powers are zero so try $f = 1, x^2$ (drop 2 h factor)

$$\begin{aligned} 2 &= c_{-1}+c_0+c_1 \\ (2/3)h^3 &= h^3(c_{-1}+c_1) \end{aligned} \tag{23}$$

3.1 Gaussian Integration

We see that adding points lets us get exact results for higher order polynomials. Suppose you are a young Gauss (1777-1855) and you have stumbled on a really neat idea, that you just know will be called "Gaussian Quadrature" one day (as if you wouldn't have enough ideas in mathematics and computing named after you already). Gauss reminds himself that to improved the approximation of integrals he uses the form

$$\int_{-1}^{1} f(x)dx \simeq \sum_{i=1}^{N} w_i f(x_i) \tag{24}$$

for $N = 1, 2, 3$, with the novelty that he lets **both** w_i **and** x_i be freely adjusted to improve the accuracy. This allows $2N$ knobes to give a perfect answer to all the powers: $1, x, x^2, \cdots, x^{2N-1}$ – a terrific idea for smooth functions! Indeed this is known, in general, as "Gaussian integration," or more appropriately "Gaussian quadrature" (see https://en.wikipedia.org/wiki/Gaussian_quadrature).

Skimming through the literature, you noticed that Legendre (1752-1833) invented a set of polynomials $P_N(x)$ of increasing order N (i.e. maximun power is x^N), whose zeroes just happend to correspond to the values x_i Gauss found in his Quadrature for $N = 1, 2, 3$. It's time to be bold: check if this works for $N = 5$ and $N = 7$. Of course, you know it's not easy to find the zeroes of $P_3(x)$, $P_5(x)$, and $P_7(x)$. A brilliant young French guy who goes by Galois (1811-1832) who was killed in a duel at age 22, proved that there is no elementary solution to the quintic[1]. Even the cubic is not easy (see https://en.wikipedia.org/wiki/Cubic_function).

Figure 4: On left Gauss of course. On right Galois – whom lost a dual at age 22!

As a quick aside: thus far, we've only been concerning ourselves with intervals $-h < x < h$, or $-1 < x < 1$. This is not an issue: we can transform any arbitrary interval $a < x < b$ into $-1 < x < 1$ by the rescaling:

$$\int_{-a}^{b} f(x')dx' = \frac{(b-a)}{2} \int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right) dx \tag{25}$$

Going forward, we'll only talk about integrating over the interval $-1 < x < 1$. Let's try our idea of Gaussian quadrature for $N = 1$ terms: (ie. 1 to x)

$$I_1 = w_0 f(0) \tag{26}$$

so

$$2 = w_0 \quad \text{for 1} \tag{27}$$

This is also known as midpoint integration:

$$I_1 = 2f(0), \tag{28}$$

where we just approximate the integral as the value of the function at the center of the interval multiplied by the width of the integral.

[1]Though due to the form of the functions, you can actually analytically find the zeroes of all the Legendre polynomials up to $P_9(x)$)

Let's now try it for $N = 2$ terms, that is, two points and two weights (ie. 1 to x^3)

$$I_2 = w_1 f(-\delta) + w_1 f(\delta) \tag{29}$$

where I used symmetry to get rid of x, x^3 by fixing x_1, x_2 to be equal in magnitude but opposite in sign and w_1, w_2 to be equal. This gives us:

$$\begin{aligned} 2 &= w_1 + w_1 \quad \text{for } 1 \\ 2/3 &= w_1\delta^2 + w_1\delta^2 \quad \text{for } x^2 \end{aligned} \tag{30}$$

This gives us $w_1 = 1$ and $w_1\delta^2 = 1/3$, $\delta = 1/\sqrt{3} = 0.57735$, Let's try it for $N = 3$ terms (i.e. 1 to x^5)

$$I_3 = w_1 f(-\delta) + w_0 f(0) + w_1 f(\delta) \tag{31}$$

where I used symmetry to get rid of x, x^3, x^5 fixing x_1, x_2, x_3 to one constants and w_i to 2

$$\begin{aligned} 2 &= w_1 + w_0 + w_1 \quad \text{for } 1 \\ 2/3 &= w_1\delta^2 + w_1\delta^2 \quad \text{for } x^2 \\ 2/5 &= w_1\delta^4 + w_1\delta^4 \quad \text{for } x^4 \end{aligned} \tag{32}$$

so $w_1\delta^2 = 1/3$, $w_1\delta^4 = 1/5$ so $\delta^2 = 3/5$ and $w_1 = 5/9, w_0 = 8/9$ and therefore

$$\int_{-1}^{1} f(x)dx \simeq \frac{1}{9}[5f(-\sqrt{3/5}) + 8f(0) + 5f(\sqrt{3/5})] \tag{33}$$

or

$$\begin{aligned} \int_{-a}^{b} f(x')dx' &= \frac{(b-a)}{18}\left[5f\left(-\frac{(b-a)}{2}\sqrt{3/5} + \frac{(b+a)}{2}\right)\right. \\ &+ \left. 8f\left(\frac{(b+a)}{2}\right) + 5f\left(\frac{(b-a)}{2}\sqrt{3/5} + \frac{(b+a)}{2}\right)\right] \end{aligned} \tag{34}$$

Magic a la Gauss. Who else! General formulas can be found online easily, for example, at: https://pomax.github.io/bezierinfo/legendre-gauss.html.

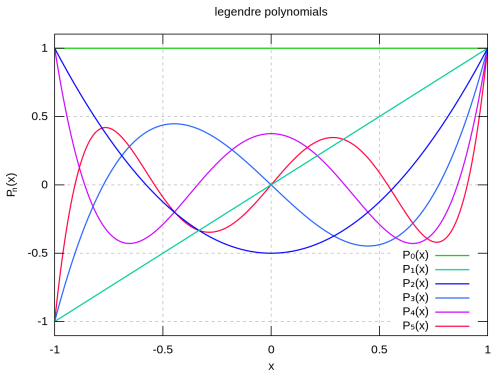
In class we discussed *Legendre* polynomials. The first few Legendre polynomials are:

$$\begin{aligned} P_1 &= x \\ P_2 &= \frac{1}{2}(3x^2 - 1) \\ P_3 &= \frac{1}{2}(5x^3 - 3x) \end{aligned}$$

As an interesting observation, the roots of these polynomials are exactly the positions where functions are sampled in I_1, I_2, and I_3 above. This generalizes for higher order Gauss-Legendre quadrature. The roots are always the position in the Gauss Lagrange methods, and the weights are given by:

$$w_i = \frac{2(1 - x_i^2)}{[(n+1)P_{n+1}(x_i)]^2} \tag{35}$$

See https://pomax.github.io/bezierinfo/legendre-gauss.html for roots and weights or just run the Mathematica program on the page.



n	$P_n(x)$
0	1
1	x
2	$\frac{1}{2}(3x^2-1)$
3	$\frac{1}{2}(5x^3-3x)$
4	$\frac{1}{8}(35x^4-30x^2+3)$
5	$\frac{1}{8}(63x^5-70x^3+15x)$
6	$\frac{1}{16}(231x^6-315x^4+105x^2-5)$
7	$\frac{1}{16}(429x^7-693x^5+315x^3-35x)$
8	$\frac{1}{128}(6435x^8-12012x^6+6930x^4-1260x^2+35)$
9	$\frac{1}{128}(12155x^9-25740x^7+18018x^5-4620x^3+315x)$
10	$\frac{1}{256}(46189x^{10}-109395x^8+90090x^6-30030x^4+3465x^2-63)$

Figure 5: On left, the plots of the Lengredre Polynomial listed on the right.

4 Lecture Notes: Root Finding

In **Numerical Recipes in C** at http://apps.nrbook.com/c/index.html in Chapter 9 there a nice dicussions on Root Finding. For Trapezoidal and Simpson's rule of integration and Sec 4.7 to find zeros of Polynomial and Sec 3.1 for Lagrangian Interpolation.

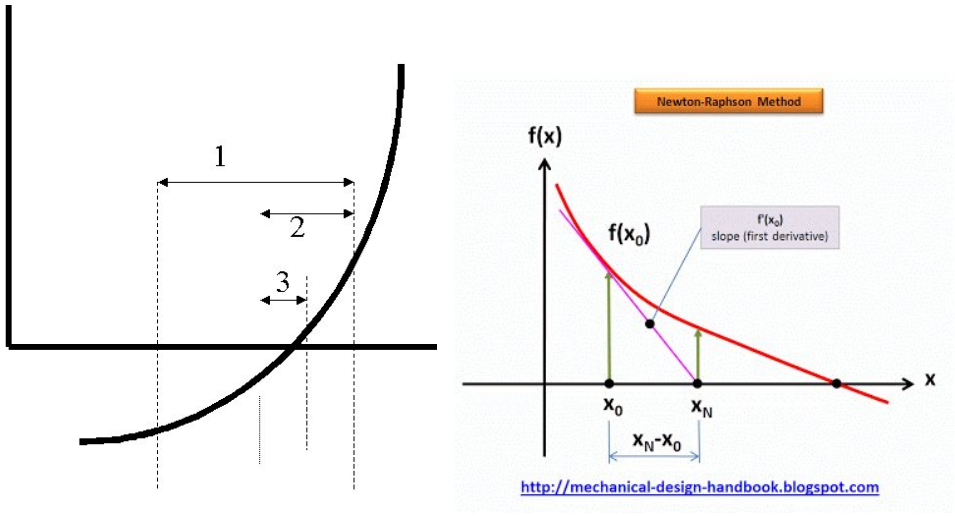


Figure 6: On left root finding by bisection. On the right root finding by Netwon-Raphson.

To find a root of a function you start with an interval $[x_{min}, x_{max}]$ where the function changes sign. For example, say $f(x_{min}) < 0$ and $f(x_{max}) > 0$, though we can switch this without too much effort. If the function is continuous (doesn't have any jumps in it, compare with the step function which does), there must be at least one zero between x_{min} and x_{max}: just one if the function monotonically increases/decreases, or more if the function oscillates

One way to find the zero is to bisect the interval repeatedly. To do this, look at $f\left(x_c \equiv \frac{x_{min}+x_{max}}{2}\right)$. If $f(x_c) < 0$, then a zero must be between $f(x_c)$ and $f(x_{max})$, so we replace our old x_{min} with x_c and repeat. If $f(x_c) > 0$, then a zero must be between $f(x_{min})$ and $f(x_c)$, so we replace our old x_{max} with x_c instead. By repeating this process, we can constrain a zero of the function $f(x)$ arbitrarily well.

Newton had a faster idea (smart computer scientist that Isaac). Start with a guess for the zero, x_0, approximate the function $f(x)$ as a straight line, and use the x intercept of that line, denoted x_1, as a new guess for the zero, then repeat.

More explicitly, we use a first order series for f(x),

$$f(x) \simeq f(x_0) + (x - x_0)f'(x_0), \tag{36}$$

and solve for the zero to get $x_1 = x_0 - f(x_0)/f'(x_0)$, or on the computer iterate

$$x \leftarrow x - f(x)/f'(x). \tag{37}$$

The classic example is to find the square root of a number A by finding the positive zero of $f(x) = x^2 - A$ using $f'(x) = 2x$. This gives us:

$$x - f(x)/f'(x) = x - \frac{x^2 - A}{2x} = \frac{x + x/A}{2} \to x$$

so the computer does the iteration,

$$x = (1/2) \times (x + x/A); \tag{38}$$

The problem set will show that this is fantastically fast relative to bisection... unless the method fails. That's life in the fastlane!

Assignment 3: Gaussian Elmination and Integration
due: Feb 13, 2017 – 11:59 PM

GOAL: After laying the groundwork, it's now time to integrate functions in Assignment #3. Over the past two weeks of class, we've covered the last necessary topic, `Gaussian Elimination`, to construct arbitrarily accurate Gaussian quadratures.

This problem set is split into two parts.

In the first part, you will write your own code to perform Gaussian elimination, extended to include row pivoting to help with numerical stability. After a few warm up examples to test your code, you'll use it to find the weights for arbitrary order Gaussian elimination. (This depends on the function `int getLegendreZero(double* zero, double* a, int n)` from the last problem set, if you couldn't complete that problem please let us know! Of course the zeros are given on the web so you can always get them to be able to proceed this way.)

In the second part, you'll put all of the pieces together and write your own numerical integration code. You'll integrate a few functions we supply on the interval $[-1, 1]$ using the trapezoidal rule, Simpson's rule, and a few different orders of Gaussian integration—this last part depends on the first part of this problem set!

Due Monday Feb 13, 2017 by 11:59PM

The code and output data and graphs are due by the start of class on Wednesday Feb 24, 2016. They should be mailed to `bualghpc@gmail.com` with the subject line `[Your name] EC500 Problem Set # 3`. ALSO please append all your files in a single tar file `YourName_PS3.tar`. There are many ways to do this but here is command in a unix shell:

```
tar -cvf     CarlGauss_PS#.tar  DirectoryWithFiles
where  CarlGauss           =  YourName
       PS#                 =  PS3  -- this time
       DirectoryWithFiles  =  contains ONLY the files you should turn in
```

II.1 Coding Exercise #1: Gaussian Eliminate all other factors, and the one which remains must be the truth.

Gaussian elimination is a general algorithm to systematically solve systems of equations. In Sec. **??** of the Lecture notes above, we gave a blow by blow example for 3 unknows:

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 2x + y + 3z &= 0 \\ 4x - 3y + z &= -10 \end{aligned} \tag{1}$$

This form lends itself nicely to multi-dimensional arrays. The left hand side can be stored as a 3x3 two-dimensional array, and the right hand side can be stored as a one-dimensional array of length 3. These are commonly referred to as A and b, respectively. These operations can all be done in place in A and b (with a temporary variable for interchanging rows when you pivot). When the algorithm is done, b contains the solution!

Your task for this problem to solve a system of linear equation Gaussian elimination: The routine only needs input of a 2D array `A` and a 1D array `b`. This algorithm is done **in place**. An **in place** algorithm is very avoiding extra memory allocation and references. Namely in each step you can over write both `A` and `b` and in the end the values in `b` give the solution. An **in place** algorithm is very important for avoiding extra memory allocation and references. With **Big Data** extra copies are a killer. In the small data example you many want to save one copy of `A` and `b` outside the function to have the orginal data for future reference and testing. The function you need to program is:

```
int gaussianElimination(double** A, double* b, int dim);
```

where

- `dim` is the *dimension* of the problem, that is, the number of equations and variables.
- `A` is a `dim` by `dim` 2D array which contains the coefficients of a system of equations.
- `b` is a 1D array of length `dim` which contains the right hand side of a system of equations.

For example, for the system of equations in Eq. **??**, one might write the code to form `A` and `b`, call the gaussian elimination function, and print the results as:

```
int i;
double** A;
double* b;
int dim = 3;
A = new double*[dim];
for (i=0;i<dim;i++) { A[i] = new double[dim]; }
b = new double[dim];

A[0][0] = -1; A[0][1] =  2; A[0][2] = -5; b[0] = 17;
A[1][0] =  2; A[1][1] =  1; A[1][2] =  3; b[1] = 0;
A[2][0] =  4; A[2][1] = -3; A[2][2] =  1; b[2] = -10;
gaussian_elimination(A,b,dim);
for (i=0;i<dim;i++) { printf("%f ", b[i]); }
```

which should print out

```
1.0 4.0 -2.0
```

Of course, this should work for any other system of equations. Try, for example,

$$x + 2y + 3z + 4t = 1 \tag{2}$$
$$4x + 8y + 6z + 7t = 2 \tag{3}$$
$$7x + 8y + 10z + 11t = 3 \tag{4}$$
$$x + y + z + 5t = 4 \tag{5}$$

which will require you to pivot. Put tests for both of these systems in a file `test_gauss_elim.c`.

The deliverables for this exercise are:

- Your own code file `test_gauss_elim.c` as defined above with the function `gaussian_elimination`.

II.2 Applying Elimination to find weights in Gaussian Quadrature

See Lecture Notes in Sec. **??** for our earlier ducussion of approximating an integral from -1 to 1 by the following form:

$$\int_{-1}^{1} f(x)dx \simeq \sum_{i=1}^{N} w_i f(x_i) \tag{6}$$

We learned that we needed N points to integrate a polynomial of degree x^{2N-1} exactly. More importantly, we discussed how the points x_i, $i = 1, 2, ..., N$ exactly coincided with the zeroes of the Legendre polynomial $P_N(x)$, which we found in Problems Set #2.

As an example, remember, for $N = 1$, $P_1(x) = x$, which has one zero $x_1 = 0$. We can find the corresponding weight w_1 by looking at the constant function, or trivially x^0:

$$\int_{-1}^{1} 1dx \quad = w_1 1 \tag{7}$$

$$2 \quad = w_1 \tag{8}$$

Which tells us we can exactly integrate a polynomial of degree $x^{2N-1} = x^1$ with the form:

$$\int_{-1}^{1} f(x)dx \quad = w_1 f(x_1) = 2f(0) \tag{9}$$

For $N = 2$, $P_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$, which has zeroes $x_1 = -\sqrt{\frac{1}{3}}, x_2 = \sqrt{\frac{1}{3}}$. We can find the corresponding weights w_1, w_2 by looking at the constant function and the linear function:

$$\int_{-1}^{1} 1dx = w_1 1 + w_2 1 \qquad \Longrightarrow 2 = w_1 + w_2 \tag{10}$$

$$\int_{-1}^{1} xdx = w_1 x_1 + w_2 x_2 \qquad \Longrightarrow 0 = -\sqrt{\frac{1}{3}} w_1 + \sqrt{\frac{1}{3}} w_2 \tag{11}$$

This is a system of equations in w_1 and w_2! It can be solved to give $w_1 = w_2 = 1$.

This can be generalized for higher order Gaussian quadrature. In general, for any N, given the

zeroes $x_1, x_2, \cdots, x_N$ of $P_N(x)$, we can consider the following system of equations:

$$\int_{-1}^{1} x^0 = w_1 + w_2 + \cdots + w_N \tag{12}$$

$$\int_{-1}^{1} x^1 = x_1 w_1 + x_2 w_2 + \cdots + x_N w_N \tag{13}$$

$$\vdots \tag{14}$$

$$\int_{-1}^{1} x^{N-1} = x_1^{N-1} w_1 + x_2^{N-1} w_2 + \cdots + x_N^{N-1} w_N \tag{15}$$

where

$$\int_{-1}^{1} x^k = \begin{cases} \frac{2}{k+1}, & k \text{ even} \\ 0, & k \text{ odd} \end{cases} \tag{16}$$

You will write a program `guass_quad_weight.c` which, given a value `N` that you ask the user for, prints out the values of the zeroes x_i and the weights w_i. You should reuse the program `getZeros.c` from the previous assignment to find the zeroes, x_i, of the Legendre polynomial $P_N(x)$. You can then set up the system of equations above and use the function `gaussianElimination` that you wrote in the previous part of the problem to compute the weights, w_i.

As an example of how the code may work, let's consider asking for the zeroes and weights for $N = 3$. User input is given on lines beginning with >.

```
> ./gauss_quad_weight
What value of N?
> 3
The zeroes and weights are given by:
x_i                 w_i
-0.774596669241483 +0.888888888888889
+0.000000000000000 +0.555555555555559
+0.774596669241483 +0.888888888888889
```

You can assume we will never ask for `N` larger than 20. You should take it upon yourself to check your answers against Wikipedia: `https://en.wikipedia.org/wiki/Gaussian_quadrature#Gauss.E2.80.93Legendre_quadrature`.

The deliverables for this exercise are:

- Your own code file `gauss_quad_weight.c` as defined above. Be sure to print the zeroes and weights with at least 15 digits of precision. Your code should make use of the following functions from previous problems:
  - `getLegendreCoeff`
  - `getLegendreZero`
  - `gaussianElimination`

II.3 Coding Exercise #2: Integrate a few Function

This problem set is a little top-heavy: with all of the work you've done, it's now time to put the pieces to integrate a few functions on the interval $[-1, 1]$. This depends heavily on the code you wrote in the previous problem, so start early and ask questions if you have them!! In this problem just write a main progam `test_integrate.c` that performs the following three integrals using the Trapezoidal rule, Simpson's rule, and most importantly Gaussian integration from $N = 2$ to 10.

$$\begin{aligned} \int_{-1}^{1} x^4 \, dx &= ? \\ \int_{-1}^{1} \cos(10x) \, dx &= ? \\ \int_{-1}^{1} \frac{1}{x^2 + 0.01} \, dx &= ? \end{aligned} \tag{17}$$

You may want to try other functions, but only for your own enjoyment.

Remember:

- With the Trapezoidal rule, you sample at two points: -1 and 1. In this case, $h = 2$. (Don't be surprised when the answers are extremely poor!!)
- With Simpson's rule, you sample at three points: -1, 0, and 1. In this case, $h = 1$.
- With Gaussian quadrature, you sample at N points given by the zeroes of $P_N(x)$ and the appropriate weights.

Be sure to print the result for each method with at least 15 digits of precision.

While you are required to submit the code, the important deliverable is a plot of the relative error between the Gaussian quadrature appoximate integral and the "exact" answer given by Mathematica:

- `N[Integrate[x^4, {x, -1, 1}],15]`
- `N[Integrate[Cos[10 x], {x, -1, 1}],15]`
- `N[Integrate[1/(x^2 + 0.01), {x, -1, 1}],15]`

as a function of the number of points N. (Don't confuse this N with the `N` in the Mathematica commands—the latter forces Mathematica to print a numberical solution!) Remember, the relative error is defined as (exact − approximate)/exact.

See the Mathematica Notebook on github that does this for $f(x) = \sin(x)$ on the interval $x \in [0, 1]$. You can play with this BUT in this problem you must submit your own C code that does these integrals. Any hack that gets the right anwser is acceptable. We will put together a more general integrator in Problem Set #4 (see comment at end of problem set).

The deliverables for this exercise are:

- Your own code file `test_integrate.c` as defined above: give the integrals from -1 to 1 of x^4, $\cos(10x)$, $1/(x^2+0.01)$ using the Trapezoidal rule, Simpson's rule, and Gaussian quadrature with $N = 2$ to 10. Be sure to print the integrals with 15 digits of precision. Your code should make use of the following functions from previous problems:
  - `getLegendreCoeff`
  - `getLegendreZero`
  - `gaussianElimination`
- Three plots which plot the relative error of integration with Gaussian quadrature against the exact answer as a function of the level of Gaussian quadrature, N. Feel free to use `gnuplot` or Mathematica to make the plots. Be sure in include labels on the axes. Use the naming convention:
  - `x4err.pdf` for the integral of x^4.
  - `cos10xerr.pdf` for the integral of $\cos(10x)$.
  - `x2p001inverr.pdf` for the integral of $1/(x^2+0.01)$.

5 Lecture Notes: Solving linear equation by Gaussian Elimination

The subject of solving system of linear equation is a very big subject. In fact maybe **the** central issue for high performance computing. Here we begin with the light (and very useful) version we need to complete the Gaussian Integration program that is needed in the next problem set. A nice note on this is in WIkipedia of course: https://en.wikipedia.org/wiki/Gaussian_elimination

As an illustration let's consider a system of equations mercilessly stolen from the internet[2]:

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 2x + y + 3z &= 0 \\ 4x - 3y + z &= -10 \end{aligned} \tag{39}$$

There are many ways to solve this system of equations. A *systematic* method to solve this system of equations is to use the *first* equation to eliminate the *first* variable from all subsequent equations. This leaves two equations in two variables, and we can recurse this process. We emphasize this as a *systematic* method for a reason: systematic methods translate well into algorithms (though, of course, not always good ones). Let's step through this explicitly for the system above.

Elimination works by subtracting one equation from another, rescaled by an appropriate factor. As a more explict example, consider the first two equations above. Let's say we wanted to eliminate x from the second equation. To do this, we could take the first equation, multiply the *entire* equation by -2 (because -1 times -2 equals 2, the coefficient of x in the second equation), then subtract it from the second equation (removing the x dependence in the second equation). To eliminate x from the third equation, we would multiply the first equation by -4. Doing this, we get

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 5y - 7z &= 34 \\ 5y - 19z &= 58 \end{aligned} \tag{40}$$

Good: the bottom two lines are now two equations in two variables.

Next, let's use the second equation to eliminate y from the third equation. We have a simple case here: this requires multiplying the second equation by 1, then subtracting it from the third equation. In general it won't be this easy! Doing this, we get:

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 5y - 7z &= 34 \\ -12z &= 24 \end{aligned} \tag{41}$$

Good! We went from having two equations in two variables to one equation in one variable on the bottom row. This is easy: we can solve this trivially for z. Now, something nice happens. Let's solve for z, plug it into the other two equations, and rearrange:

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 5y - 7z &= 34 \\ z &= -2 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} -x + 2y + 10 &= 17 \\ 5y + 14 &= 34 \\ z &= -2 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} -x + 2y &= 7 \\ 5y &= 20 \\ z &= -2 \end{aligned} \tag{42}$$

The bottom equation is now trivial, and look at the top two lines: it's two equations in two variables, but now the second equation is easy to solve! This is the idea of back substitution. Carrying on, we get:

$$\begin{aligned} -x + 2y &= 7 \\ y &= 4 \\ z &= -2 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} -x + 8 &= 7 \\ y &= 4 \\ z &= -2 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} -x &= -1 \\ y &= 4 \\ z &= -2 \end{aligned} \tag{43}$$

[2] http://www.sparknotes.com/math/algebra2/systemsofthreeequations/section1.rhtml

And last but not least, the top line is easy to solve, giving us the final answer:

$$\begin{aligned} x &= 1 \\ y &= 4 \\ z &= -2 \end{aligned} \tag{44}$$

And we're done!

Before we get into transforming this into something more suitable, let's consider one special case. What if you were given the system

$$\begin{aligned} 5y - 7z &= 34 \\ -x + 2y - 5z &= 17 \\ 4x - 3y + z &= -10 \end{aligned} \tag{45}$$

If you followed my steps above very naively, you'd run into an issue: there's no term with x in the top equation! There's nothing I can multiply the first equation by to eliminate x from the second and third equation. Of course, as a human, you know easy ways to address this. One method is to just reorder the equations so the top equation has a non-zero coefficient of x, say, swapping the first and third row:

$$\begin{aligned} 4x - 3y + z &= -10 \\ -x + 2y - 5z &= 17 \\ 5y - 7z &= 34 \end{aligned} \tag{46}$$

This is known as *row pivoting*, or interchanging two rows. An equation goes across in a row, after all. This will make more sense soon. For numerical reasons, it's good to *always* pivot on each step such that the value with the largest magnitude (absolute value) is in the top row. As a remark, another way to address this is to reorder the variables, say reorder x and y. This is perfectly valid, and known as *column pivoting*. We won't go into that further—it takes a bit more thinking down the line, and row pivoting works well enough.

To take a last step before writing code, we can abstract this away by writing *matrices* of the coefficients. Let's consider the original system of equations:

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 2x + y + 3z &= 0 \\ 4x - 3y + z &= -10 \end{aligned} \tag{47}$$

In the painful details I demonstrated, the only operations we did were on the coefficients of the variables x, y, and z: the variables themselves just came along for the ride. Why keep writing them. Instead, we can write the system as:

$$\begin{bmatrix} -1 & 2 & -5 & 17 \\ 2 & 1 & 3 & 0 \\ 4 & -3 & 1 & -10 \end{bmatrix} \tag{48}$$

We can follow the same steps as before, except instead of rescaling *equations* and subtracting them from each other, we rescale *rows*. Interchanging equations becomes interchanging rows. (Now the name "row pivoting" should make sense!)

These operations now follow a sequence:

$$\left(\begin{array}{ccc|c} -1 & 2 & -5 & 17 \\ 2 & 1 & 3 & 0 \\ 4 & -3 & 1 & -10 \end{array}\right) = \left(\begin{array}{ccc|c} -1 & 2 & -5 & 17 \\ 0 & 5 & -7 & 34 \\ 0 & 5 & -19 & 58 \end{array}\right)$$
$$= \left(\begin{array}{ccc|c} -1 & 2 & -5 & 17 \\ 0 & 5 & -7 & 34 \\ 0 & 0 & -12 & 24 \end{array}\right)$$
$$= \left(\begin{array}{ccc|c} -1 & 2 & -5 & 17 \\ 0 & 5 & -7 & 34 \\ 0 & 0 & 1 & -2 \end{array}\right)$$
$$= \left(\begin{array}{ccc|c} -1 & 2 & 0 & 7 \\ 0 & 5 & 0 & 20 \\ 0 & 0 & 1 & -2 \end{array}\right)$$
$$= \left(\begin{array}{ccc|c} -1 & 2 & 0 & 7 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & -2 \end{array}\right)$$
$$= \left(\begin{array}{ccc|c} -1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & -2 \end{array}\right)$$
$$= \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & -2 \end{array}\right)$$

We can now read the solution off from the rightmost column: the first variable, x, equals 1, the second variable, y, equals 4, and the third variable, z, equals -2. There are important things to note on each row. On the third row, every element below the diagonal is 0: this means we reduced the matrix to *upper triangular*. This was an important step when we were solving the equation explicitly: this is when we started back substituting. On the last line, we see something else important: the left hand side is all 1 on the diagonal, and 0 otherwise. This is known as something special: the identity matrix. When the left hand side is the identity matrix, we know we're done.