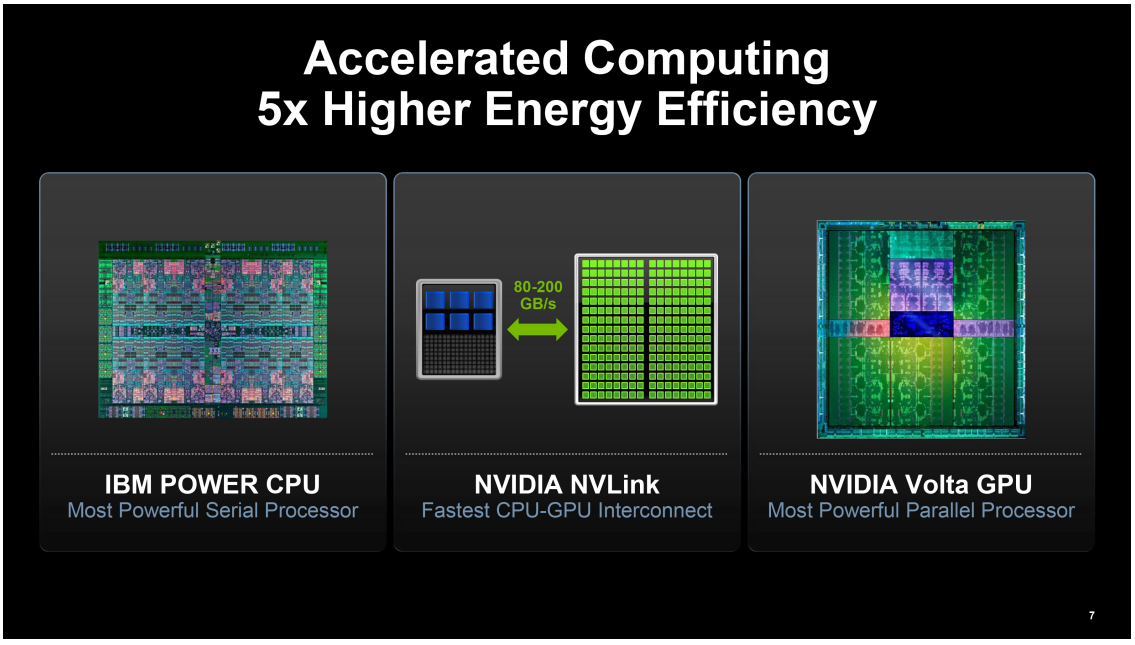
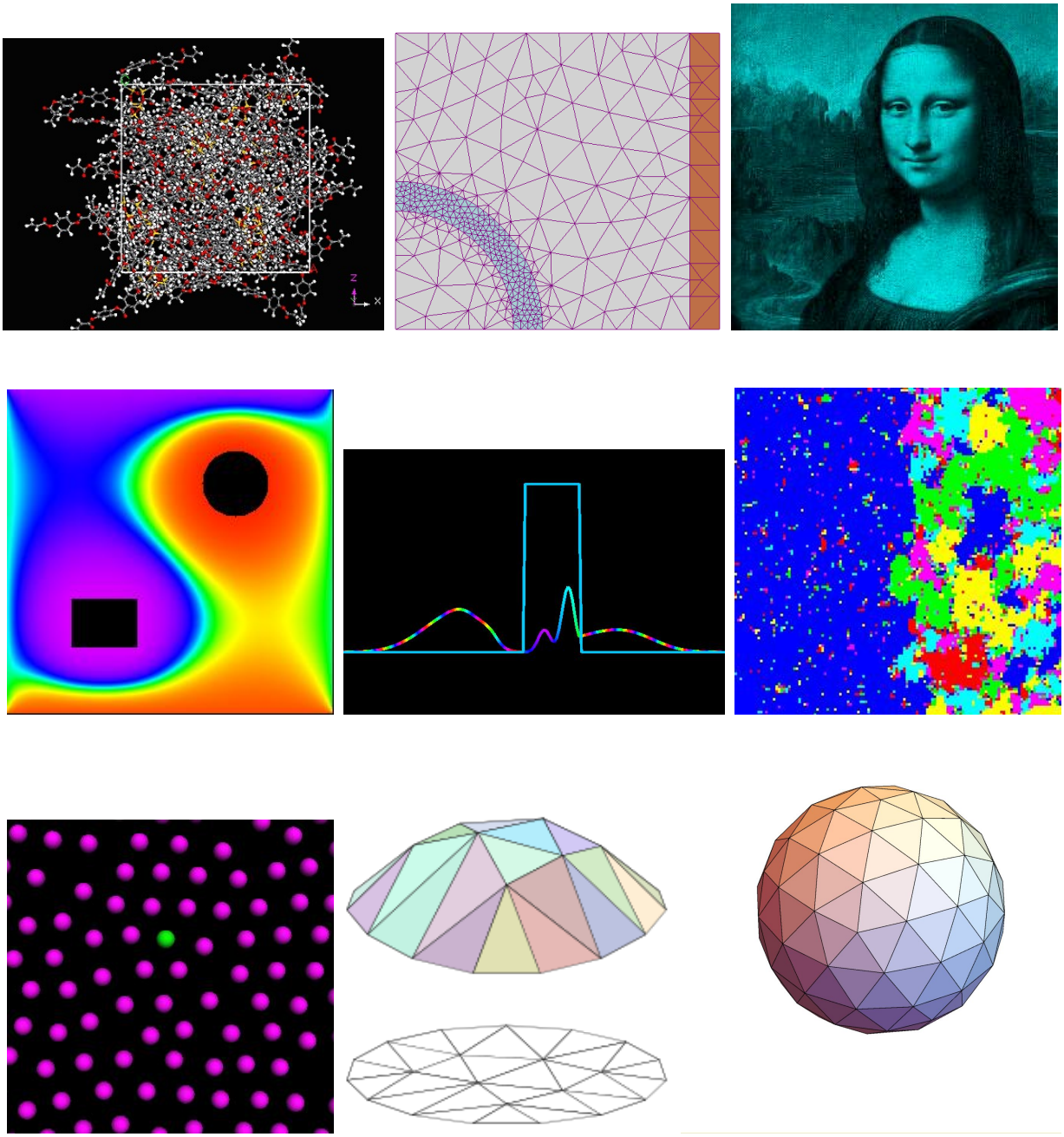
EC500 Paralllel Software for High Performance Computing

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

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

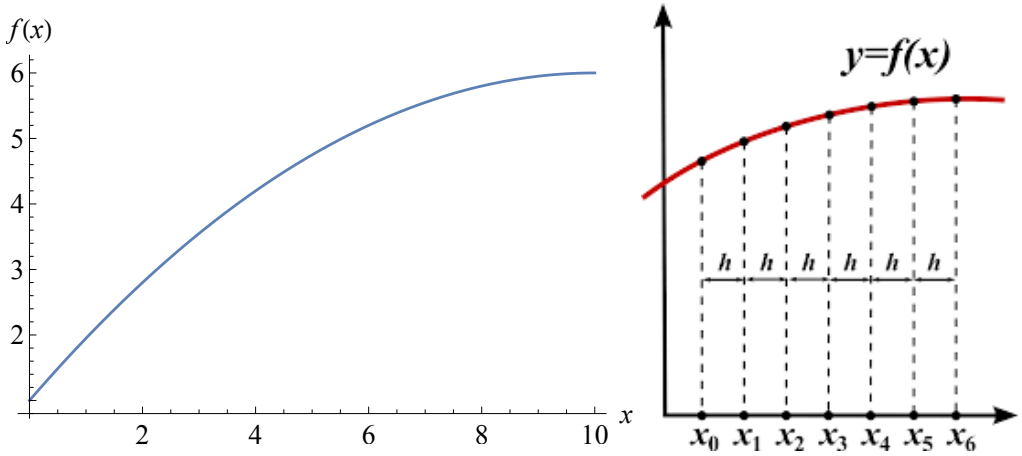


Figure 1: On left a function of x. On right the discrete points evalueted on a grid

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.

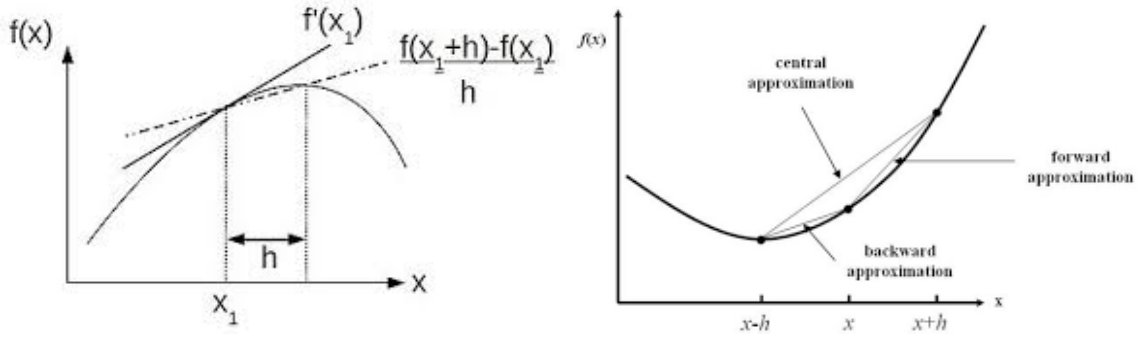


Figure 2: On left, finite difference. On right, central difference.

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 $D_h f(x)$

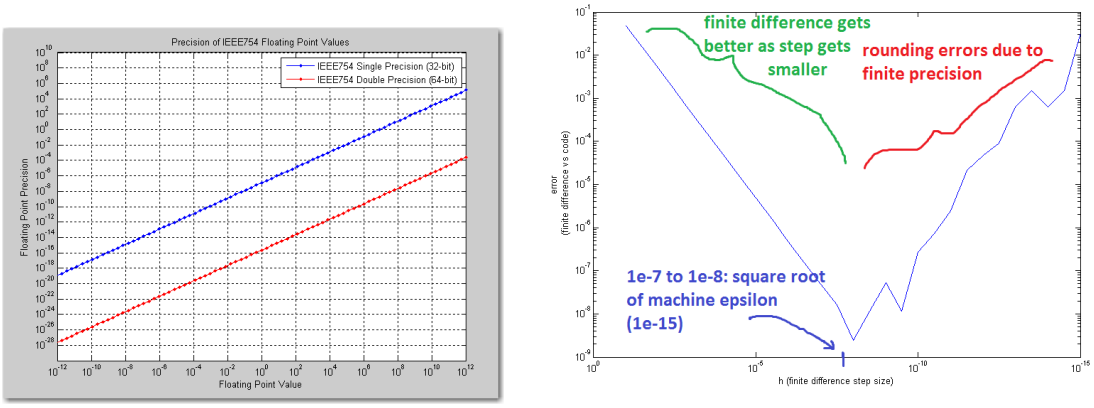


Figure 3: On the Right the error in the finite difference approximation to a deriative (y-axis) relative to size set size h (x-axis).

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

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

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,

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

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:

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

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.)

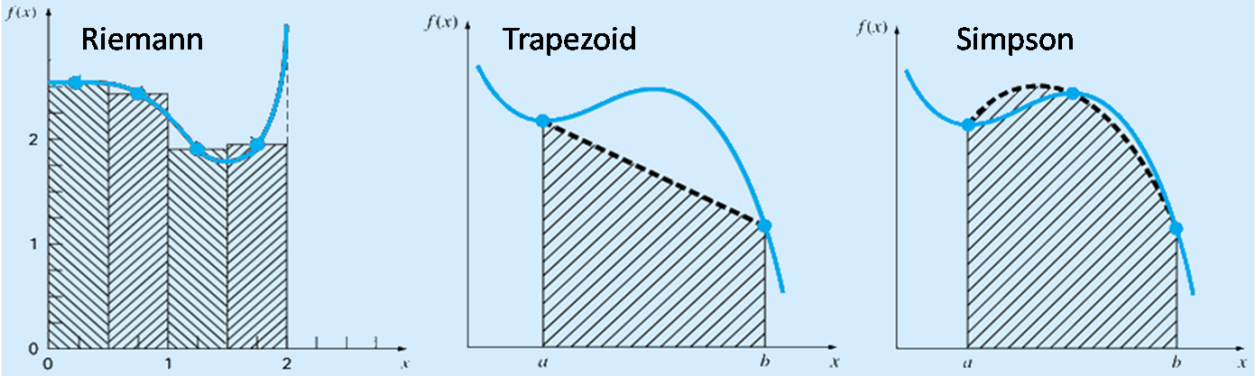
Similarly we can define $\widetilde{D}_h$ as an explicit expansion of $\widetilde{\Delta}_h$. What is? (Hint: Think what happen if $h \to -h$.) So the central difference cancels all term odd in h.

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

This is good for any finite difference $\frac{1}{2}(D_h + \widetilde{D}_h) f(x)$ 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

3 Lecture Notes: Integrals to Sums

In **In Numerical Analysis** by Burden & Fairs read Sec 4.3 for Trapezoidal and Simpson's rule of integration and Sec 4.7 on Gaussian Integration. In fact for a more straightforward approach see **Numerical Recipes in C** at http://apps.nrbook.com/c/index.html Ch 4 Sec 4.1 - 4.2 and the lecture on Gaussian integration.



The solution(s) to the equation

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

We define D^{-1} to be the integral up to an unknown constant. We can call this an "inverse Derivative"

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

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:

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

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.

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

All terms cancel except the first, if we take $f(-1) = 0$ as a convention. *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.*

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

$$I = \int_b^a f(x)dx \to I_N = \sum_{i=1}^{N} w_i f(x_i) \tag{17}$$

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 $\Delta x \equiv h = (b-a)/(N+1)$ or $x_i = a + i\Delta$ for $(x_0, x_1, ..., x_{N+1})$

$$\begin{aligned} y &= \sum_{i=0}^{N-1} h[f(a+ih) + f(a+(i+1)h)]/2 \\ &= h[\frac{1}{2}f(x_0) + f(x_1) + f(x_2) \cdots f(a+x_N) + \frac{1}{2}f(x_{N+1}) \end{aligned} \tag{18}$$

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

$$\int_{x_1}^{x_2} f(x) = h[f_1 + f_2]/2 + O(h^3 f'') \tag{19}$$

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 $c_{-1} = c_1 = 1/2$. Note x^{odd} gives zero automatically. This implies

$$y_N = \sum_{i=0}^{N-1} (x_{i+1} - x_i) \frac{f(x_i) + f(x_{i+1})}{2} \tag{20}$$

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 is good up to $O(x^3)$ exactly:

$$\int_{-h}^{h} f(x)dx = h\left[\frac{1}{3}f(-h) + \frac{4}{3}f(0) + \frac{1}{3}f(h)\right] + O(h^5 f^{(4)}) \tag{21}$$

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

$$\begin{aligned} 2 &= c_{-1} + c_0 + c_1 \\ (2/3)h^3 &= h^3(c_{-1} + c_1) \end{aligned} \tag{22}$$

3.1 Gaussian Integration

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

We see that adding points lets us get exact results for higher order polynomials. To this point, though, we've only let the weights in our approximations change. What if we let both the weights and the positions to be tuned to get a better approximation?

$$I_N = \int_{-1}^{1} f(x)dx = \sum_{i=1}^{N} w_i f(x_i) \tag{23}$$

This is known, in general, as "Gaussian integration," or more appropriately "Gaussian quadrature" (see https://en.wikipedia.org/wiki/Gaussian_quadrature). We have $2N$ variables to play with: x_i, w_i. So we can get rid of $2N$ integrals over $1, x, x^2, \cdots, x^{2N-1}$ exactly!

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:

$$\int_{-a}^{b} f(x')dx' = \frac{(b-a)}{2} \int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right) dx \tag{24}$$

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)

$$I_1 = w_0 f(0) \tag{25}$$

so

$$2 = w_0 \quad \text{for } 1 \tag{26}$$

This is also known as midpoint integration:

$$I_1 = 2f(0), \tag{27}$$

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.

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

$$I_2 = w_1 f(-\delta) + w_1 f(\delta) \tag{28}$$

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:

$$\begin{aligned} 2 &= w_1 + w_1 \quad \text{for } 1 \\ 2/3 &= w_1\delta^2 + w_1\delta^2 \quad \text{for } x^2 \end{aligned} \tag{29}$$

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)

$$I_3 = w_1 f(-\delta) + w_0 f(0) + w_1 f(\delta) \tag{30}$$

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

$$\begin{aligned} 2 &= w_1 + w_0 + w_1 \quad \text{for } 1 \\ 2/3 &= w_1\delta^2 + w_1\delta^2 \quad \text{for } x^2 \\ 2/5 &= w_1\delta^4 + w_1\delta^4 \quad \text{for } x^4 \end{aligned} \tag{31}$$

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

$$\int_{-1}^{1} f(x)dx \simeq \frac{1}{9}[5f(-\sqrt{3/5}) + 8f(0) + 5f(\sqrt{3/5})] \tag{32}$$

or

$$\begin{aligned} \int_{-a}^{b} f(x')dx' &= \frac{(b-a)}{18}\left[5f\left(-\frac{(b-a)}{2}\sqrt{3/5} + \frac{(b+a)}{2}\right)\right. \\ &+ \left. 8f\left(\frac{(b+a)}{2}\right) + 5f\left(\frac{(b-a)}{2}\sqrt{3/5} + \frac{(b+a)}{2}\right)\right] \end{aligned} \tag{33}$$

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

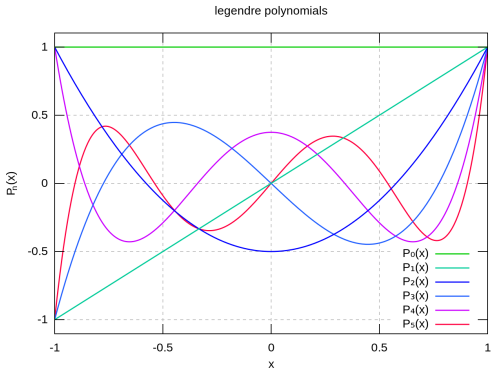
In class we discussed *Legendre* polynomials. The first few Legendre polynomials are:

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

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:

$$w_i = \frac{2(1 - x_i^2)}{[(n+1)P_{n+1}(x_i)]^2} \tag{34}$$

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: Parallelization Methods

The parallelization of large scale high performance code is essential. This will be done by using the Shared Computer Cluster (SCC) at the Massachusetts Green High Performance Computer Center (MGHPCC). The MGHPCC operates as a joint venture between Boston University, Harvard University, the Massachusetts Institute of Technology, Northeastern University, and the University of Massachusetts. http://www.mghpcc.org

4.1 Accessing SCC

To access `scc`, open a terminal and run:

```
ssh ~[username]@scc1.bu.edu
```

My default shell on `scc` isn't bash, so I normally run the command `bash` to switch over to a bash shell. Before running `bash`, the command prompt looks like:

```
scc1:~ %
```

On the other hand, the default bash command prompt includes your username:

```
[username]@scc1 \textasciitilde]
```

Good! Next, let's navigate to our project's active directory.

```
cd /projectnb/isingmag/
```

You should create your own directory with your kerberos username. If you'd like to play around with compiling and running code, you can log into an interactive shell by running:

```
qrsh -l h_rt=1:00:00 -P isingmag
```

That'll give you a one core interactive shell for one hour. Of course, it should be pretty clear how to try to get it for longer than an hour... but keep in mind, a longer interactive shell may not be available! There's a wonderful collection of information on BU's IS&T website for running jobs on SCC, if you're interested: http://www.bu.edu/tech/support/research/system-usage/running-jobs/

4.2 Hello World

Let's start with a simple hello world code!

```
#include <stdio.h>

int main(int argc, char** argv)
{
   printf("Hello world!\n");
   return 0;
}
```

Let's say we saved this to a file `hello.c`. It could be compiled by:

```
gcc hello.c -o hello
```

Okay, good, we did the obvious.

4.3 OpenMP

Let's consider a parallel version using OpenMP:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv)
{
   int tid, nthreads;

   #pragma omp parallel shared(nthreads) private(tid)
   {
      tid = omp_get_thread_num();
      if (tid == 0)
      {
         nthreads = omp_get_num_threads();
      }

      #pragma omp barrier

      printf("Hello world from thread %d of %d!\n", tid, nthreads);
   }
   return 0;
}
```

Which can be compiled by:

```
gcc omp_hello.c -fopenmp -o omp_hello
```

Note the additional `include` and the added compile flag `-fopenmp`. There's more—we run this code by first defining the number of threads to use. This can be done by a bash environment variable:

```
export OMP_NUM_THREADS=4;
./omp_hello
```

This would be the case if we were on a 4 core machine. By the way, we would get a parallel interactive thread by:

```
qrsh -l h_rt=1:00:00 -pe omp 4 -P isingmag
```

OpenMP works largely through the `#pragma` statements. The line:

```
#pragma omp parallel shared(nthreads) private(tid)
```

instructs the compiler to run that code in parallel on each core. The pragma explicitly calls out two variables, `nthreads` and `tid`, as shared and private, respectively. Each thread has its own `tid`, which makes sense—it's the numerical id of the thread! On the other hand, the number of threads will be same in every thread. Thus, we mark `nthreads` as shared.

More to prove a point, we have only the first thread (0) calculate the number of threads. That said, since it's in shared memory, we need to wait for all threads to sync up. That's what the `barrier` is for. After the barrier, we have every thread print its name, and carry on! Last, let's turn to multi-node parallelism through MPI (stands for Message Passing Interface).

```
// Based on the tutorial from mpitutorial.com/tutorials/mpi-hello-world/

#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv)
{
   int world_size; // number of MPI processes
   int world_rank; // rank of MPI process
   char processor_name[MPI_MAX_PROCESSOR_NAME]; // name of processor
   int name_len; // length of name string.

   // Initialize MPI
   MPI_Init(&argc, &argv);

   // Get the number of processes
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   // Get the rank
   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

   // Get the name of the processor
   MPI_Get_processor_name(processor_name, &name_len);

   printf("Hello world from rank %d of %d! My name is %s!\n",
                        world_rank, world_size, processor_name);

   // Clean up
   MPI_Finalize();
   return 0;
}
```

You'll notice there are no pragmas here. Similar to the OpenMP code, though, we can learn the total number of processes (`MPI_Comm_size`, analagous to `omp_get_num_threads`), and the id of the current process (`MPI_Comm_rank`, analagous to `omp_get_thread_num`). You'll notice there's no barrier here, but the functionality is there. MPI runs more appropriately with Send/Receive calls, which we'll get to soon.

First, we need to discuss how to compile and run MPI code. Instead of using `gcc`, we use `mpicc`:

```
mpicc mpi_hello.c -o mpi_hello
```

And you don't just "run" an mpi program; you start it with `mpirun`:

```
mpirun -np 2 ./mpi_hello
```

The flag `-np 2` means to run the program on 2 processes. You can try this on the log-in node, even! (Up to 4 processes, for a short time.)

4.4 Message Passing

Parallelizing with OpenMP can be decently straightforward, in large part because of shared memory: different threads can all access the same information. This simplification breaks down when programs are being run on more than one node, as is the case with MPI programs! To share information between MPI jobs, we need to take the "Message Passing" part of MPI seriously.

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

An (admittedly unclear) example:

```
MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
```

- `&offset`: Pointer to start of buffer.
- `1`: Non-negative number of elements in buffer. (For ex, if buffer is an array of `ints`, number of ints.
- `MPI_INT`: Data type. A list of data types can be found at http://linux.die.net/man/3/mpi_int. We'll work with `MPI_INT`, `MPI_DOUBLE`.
- `dest`, `source`: The destination/source. This is the integer "rank" of the process which we found from `MPI_Comm_rank`.
- `mtype`: An index of the type of message. Could just use "0" if you have only one message type.
- `MPI_COMM_WORLD`: The communication channel. I'm not sure how we would change this—something to learn about!
- `&status`: Extra information of type `MPI_Status`.

These are *blocking commands*: they wait until the two-way communication is complete.

4.5 Submitting Jobs

OpenMP: To submit a 4 thread OMP job, run the following command:

```
qsub -pe omp 4 -v OMP_NUM_THREADS=4 -b y -l h_rt=0:05:00 -P isingmag omp_hello
```

- `-pe omp 4` specifies a 4 node job. This can be changed!
- `-v OMP_NUM_THREADS=4` passes a command line variable (implied by `-v`). We know from above this variable informs the OMP job of how many threads to use. This should be consistent with the previous command.
- `-b y` specifies that we are directly submitting a binary. We can also submit scripts, which is a better way to use batch queuing systems! We get to that later.
- `-l h_rt=0:05:00` specifies that the max time this job will take is 5 minutes. There is a default max time on SCC, depending on the type of job. This information is listed on http://www.bu.edu/tech/support/research/system-usage/running-jobs/parallel-batch/. You can change the max wall time, but remember, don't set it longer than it needs to be (do some timings)—shorter jobs may start running sooner!
- `-P isingmag` specifies to use the project allocation `isingmag`. **You must use this to run programs for this class.**
- `omp_hello` specifies the executable name. Obviously, this can change.

Information on more flags can be found at http://www.bu.edu/tech/support/research/system-usage/running-jobs/submitting-jobs/

MPI: As we mentioned before, submitting a job via a bash script is better practice than directly submitting an executable to the queueing system. I modified a simple submit script from http://www.bu.edu/tech/support/research/system-usage/running-jobs/parallel-batch/ to run our MPI hello world program from before:

```
#!/bin/sh
#
#$ -pe mpi_4_tasks_per_node 8
#
# Invoke mpirun.
# SGE sets $NSLOTS as the total number of processors (8 for this example)
#
mpirun -np $NSLOTS ./mpi_hello
```

We'll explain this in a moment. This script is submitted by:

```
qsub -l h_rt=0:05:00 -P isingmag submit_mpi_hello.sh
```

This submit command should look largely familiar! There are no arguments related to OpenMP since we aren't using it (thus the `-pe omp`, `-v` arguments are gone). We're submitting a script instead of a binary so we don't use `-b y`. Let's turn to the submit script. One line at the top should look particularly familiar:

```
#$ -pe mpi_4_tasks_per_node 8
```

I didn't say this before, but `-pe` stands for *parallel environment*. In this case, we wanted an MPI environment—that's implicit in the `mpi` part of `mpi_4_tasks_per_node`. The rest of the argument refers to how many processes, or *tasks* we want per node, or alternatively how many cores on the node to use. Here, we asked for 4 processes per node. (Other acceptable arguments are `mpi_8_tasks_per_node`, `mpi_12_tasks_per_node`, `mpi_16_tasks_per_node` for 8, 12, and 16 processes per node, respectively.)

The argument 8 specifies the total amount of processes we would like to use, which in turn implies how many nodes we want. Here, we asked for 8 processes, with 4 per node. Therefore, we're implicitly asking for 2 nodes.

If we submit the job:

```
> qsub -l h_rt=0:05:00 -P isingmag submit_mpi_hello.sh
Your job 6404924 ("submit_mpi_hello.sh") has been submitted
```

we get the following output:

```
> cat submit_mpi_hello.sh.o6404924
Hello world from rank 0 of 8! My name is scc-ja1!
Hello world from rank 1 of 8! My name is scc-ja1!
Hello world from rank 2 of 8! My name is scc-ja1!
Hello world from rank 3 of 8! My name is scc-ja1!
Hello world from rank 4 of 8! My name is scc-je2!
Hello world from rank 5 of 8! My name is scc-je2!
Hello world from rank 6 of 8! My name is scc-je2!
Hello world from rank 7 of 8! My name is scc-je2!
```

Where we can see we did use 8 processes total, and bsaed on the name we can see we ran on two different nodes!

4.6 Checking the status of a running job

You can check the status of running jobs by running the command:

```
qstat -u [username]
```

We've only scratched the surface of the options available for writing programs and submitting jobs on SCC. This document serves as a launching point. As you're interested, dig through the information on the IS&T website, do some googling, and get cracking at writing parallel code!