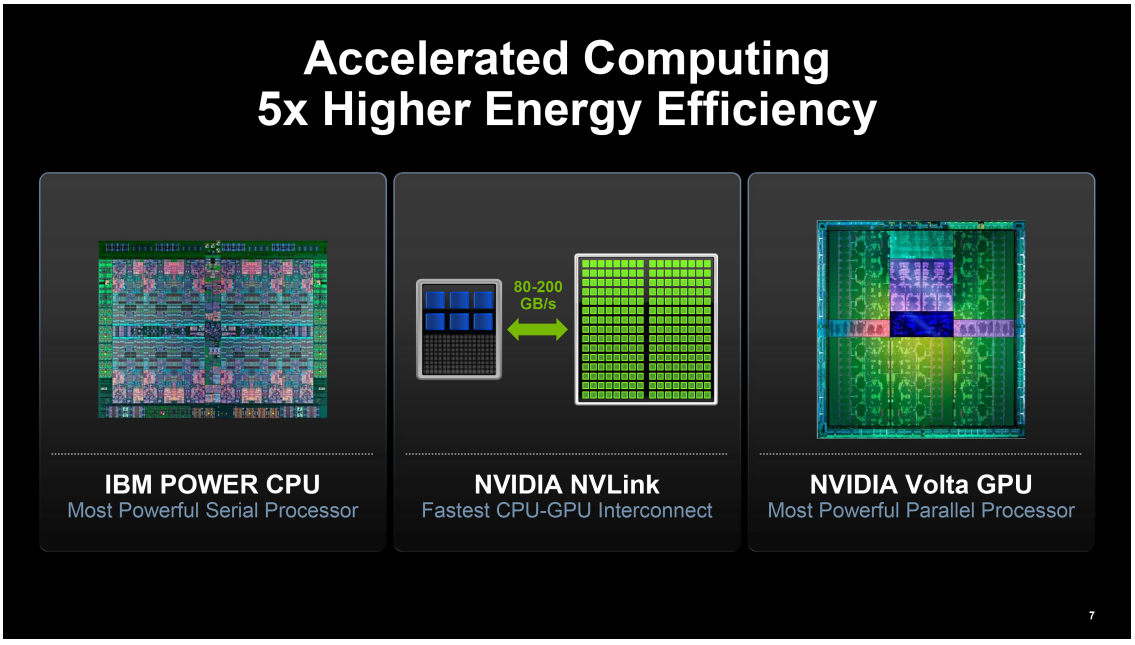
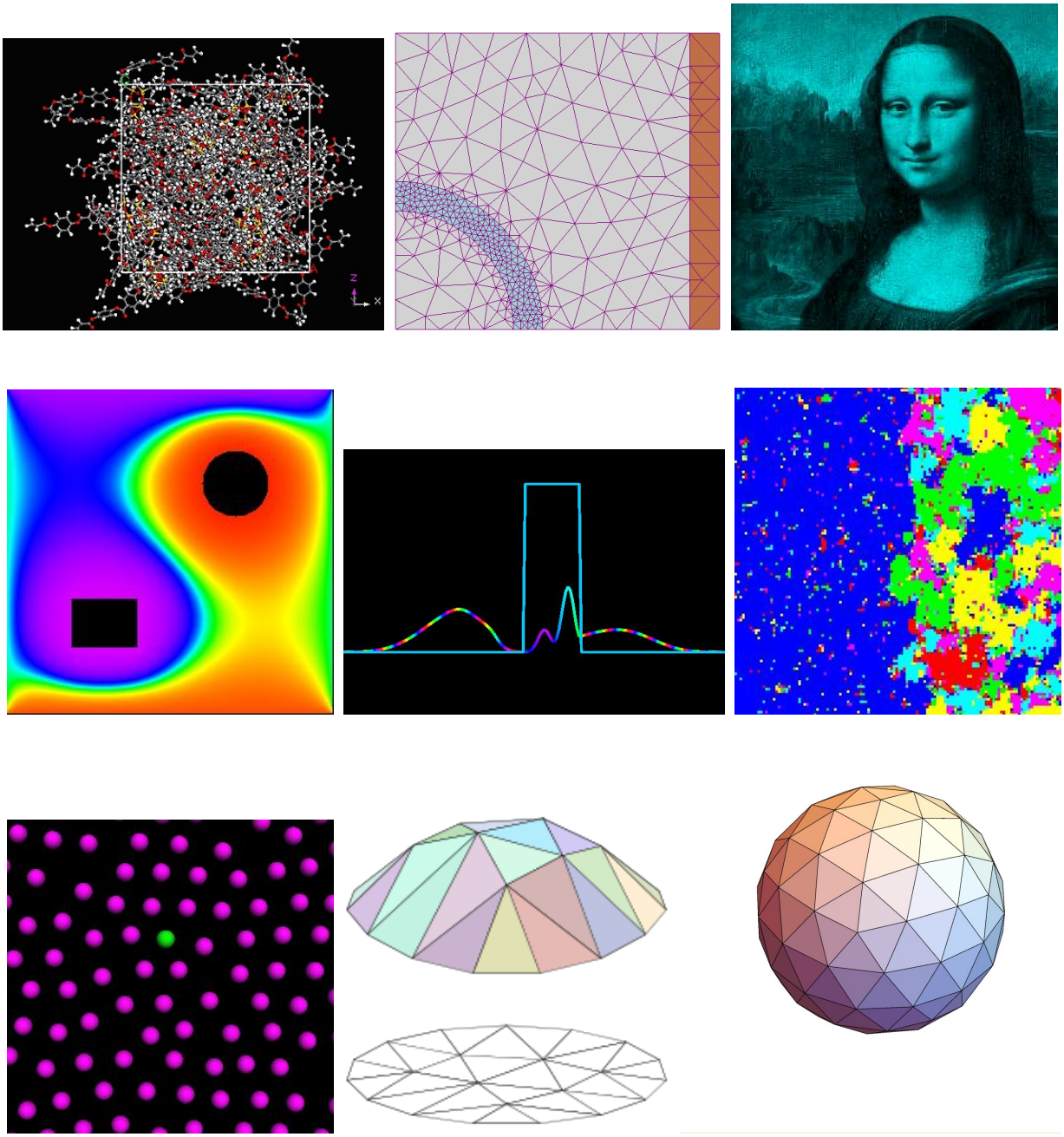
EC500 Paralllel Software for High Performance Computing

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

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 weekly HW exercies (2/3), a final project and participation in the class (1/3). You are encouraged to discuss basic concepts in each HW exercises in class and with other students BUT the actual code/data output/figures must be exclusively yours alone. Upto a week late the HW will be graded but to get full credit you must request the delay and give reasons for being overloaded. The project will include code, performance analysis and write up and a presentation in the last week. The project can be in a small team of 2 to 3 individuals.

2 Lecture Notes: Derivative to Finite Differences

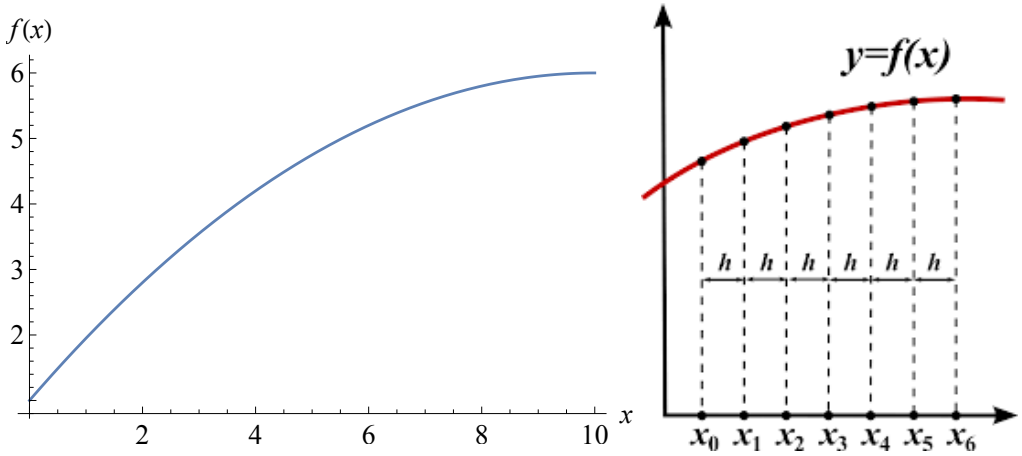


Figure 1: On left a function of x. On right the discrete points evalueted on a grid

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.

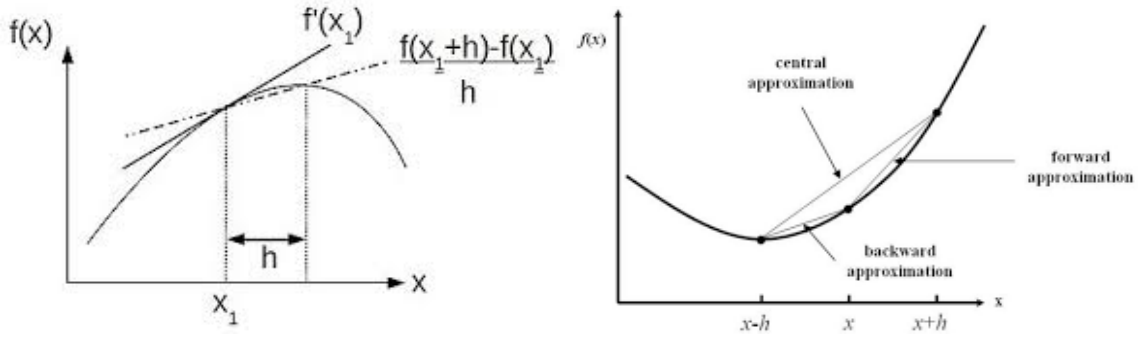


Figure 2: On left, finite difference. On right, central difference.

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 $\Delta_h f(x)$

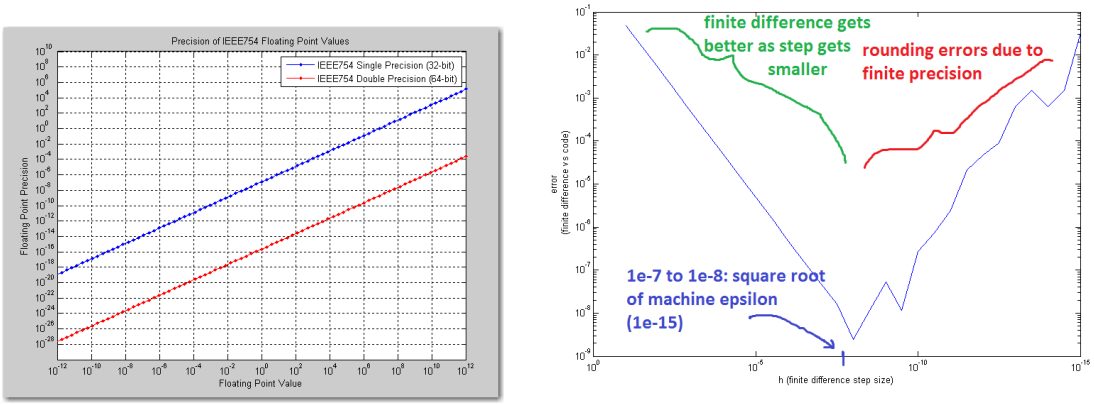


Figure 3: On the Right the error in the finite difference approximation to a deriative (y-axis) relative to size set size h (x-axis).

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.)

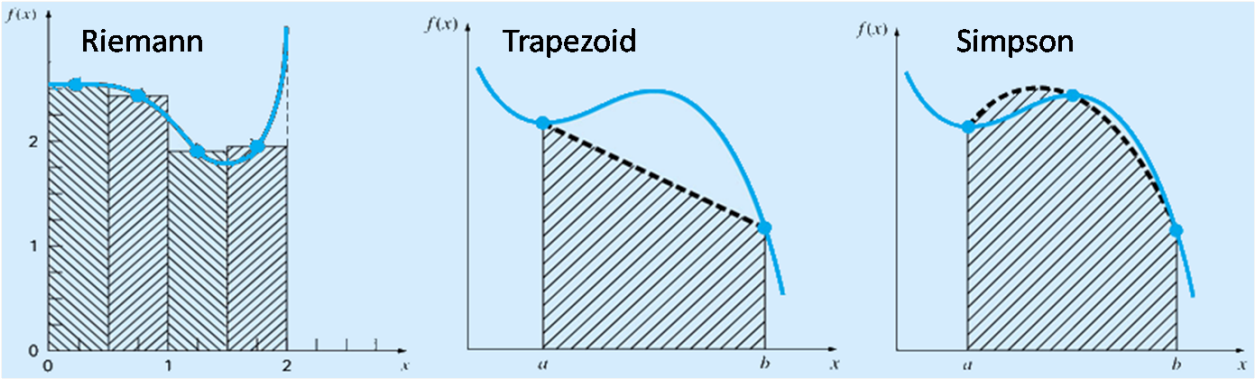
Similarly we can define $\widetilde{\Delta}_h$ as an explicit expansion of $\widetilde{\Delta}_h$. What is? (Hint: Think what happen if $h \to -h$.) So the central difference cancels all term odd in h.

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

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

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.

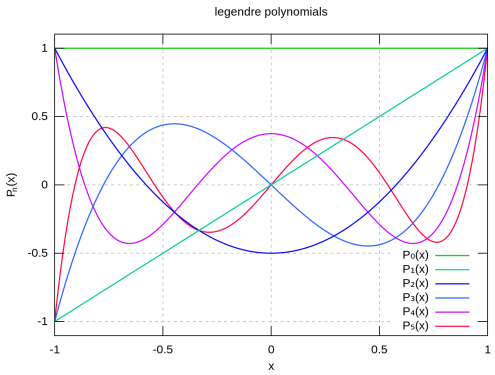
In class we discussed *Legendre* polynomials. The first few Legendre polynomials are:

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

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.

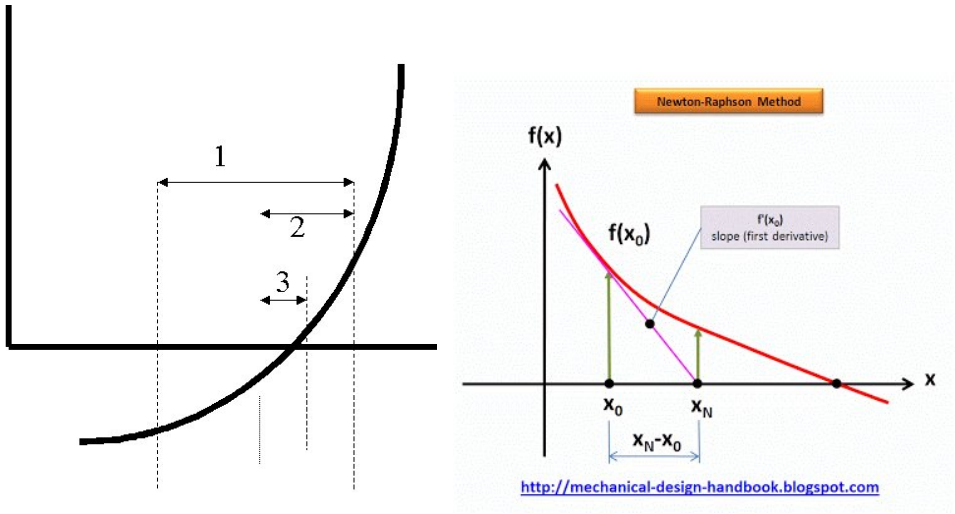


Figure 6: On left root finding by bisection. On the right root finding by Netwon-Raphson.

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!

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:

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

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

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:

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

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.

6 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

6.1 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/

6.2 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.

6.3 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.)

6.4 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.

6.5 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!

6.6 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 cOde1!

7 Lecture Notes: Curve fitting &Error analysis

As a next step, we will consider curve fitting. Later we will see that curve fitting when there is a large amount of data turns into a system of equations, connecting back to the previous topic of systems of equations and Gaussian elimination. For now, though, we will consider a simple case: fitting to a line.

Back in secondary education, you must have spent a lot of time learning how to fit two data points to a line. Two points (that aren't vertically displaced from each other) can be fit to a line in *slope-intercept* form, that is:

$$y = mx + b \tag{49}$$

where m and b are the slope and y-intercept. Given two points (x_1, y_1) and (x_2, y_2), you have a system of equations in m and b, and you can construct a unique line.

What if we could generalize this? Not only make it more constructive (not requiring solving a system of equations, for one), but also make it generalizeable to fitting an arbitrary number of points to a polynomial?

Let's motivate this by first starting with a simple example: fitting a line to two points, $(1, 2)$ and $(3, 4)$. First, just so there are no surprises: the answer is $y = x + 1$. But now let's take a more systematic approach. We can split this into two problems:

- Find a line which is 2 at $x = 1$, and 0 at $x = 3$.
- Find a line which is 4 at $x = 3$, and 0 at $x = 1$.

We can split this into two pieces like so because *the sum of two lines is a line* (and, more generally, the sum of any number of lines is a line, any number of quadratic polynomials is a quadratic polynomial, etc).

We'll focus on the first line in pieces: a line which is 2 at $x = 1$, 0 at $x = 3$. Let's call this line $L_1(x)$.

First: how to we make the line 0 at $x = 3$? This requires a killing term:

$$L_1(x) \propto (x - 3) \tag{50}$$

We can multiply $L_1(x)$ by anything (which we will do to satisfy other constraints), and it will still equal 0 at $x = 3$. Good.

Next, let's make $L_1(x)$ equal 1 at $x = 1$. Why 1? Well, we can then just multiply the entire line by 2 to get the final answer. We're just breaking it down here. To make $L_1(x)$ equal 1 at $x = 1$, we need to divide what we have thus far by $(1 - 3)$: the current form of $L_1(x)$ at $x = 1$.

$$L_1(x) \propto \frac{(x-3)}{(1-3)} \tag{51}$$

Good: we now have a line that is 1 at $x = 1$, 0 at $x = 3$. For the last step, we want it to equal 2 at $x = 1$, so we just multiply by 2:

$$L_1(x) = 2\frac{(x-3)}{(1-3)} \tag{52}$$

We now have a line which is 2 at $x = 1$, and 0 at $x = 3$.

We can repeat this for the other line: a line which is 4 at $x = 3$, 0 at $x = 1$. In order:

$$\begin{aligned} L_2(x) &\propto (x-1) \\ &\propto \frac{(x-1)}{(3-1)} \\ &= 4\frac{(x-1)}{(3-1)} \end{aligned} \tag{53}$$

Good. Now, our final answer is the sum of the two lines:

$$\begin{aligned} y &= L_1(x) + L_2(x) \\ &= 2\frac{(x-3)}{(1-3)} + 4\frac{(x-1)}{(3-1)} \end{aligned} \tag{54}$$

One can quickly verify that the line is 2 at $x = 1$, 4 at $x = 3$. We can also simplify it at get the expected result $y = x + 1$.

We can generalize this now to exactly fitting a polynomial. One can prove (it's actually just follows from a system of equations!) that $n + 1$ points (x_i, y_i), $i = 1, \cdots, n + 1$ can be exactly fit by a *degree n polynomial*, that is, a polynomial whose highest power is x^n. We can generalize our method for fitting a line (a degree 1 polynomial) to two points for this arbitrary case.

As a first step, let's consider three points: $(1, 5)$, $(9, 2)$, and $(4, 1)$. Similar to the linear case, we can split this into three problems:

- Find a curve $L_1(x)$ which is 5 at $x = 1$, 0 at $x = 9$ and 4.

- Find a curve $L_2(x)$ which is 2 at $x = 9$, 0 at $x = 1$ and 4.
- Find a curve $L_3(x)$ which is 1 at $x = 4$, 0 at $x = 1$ and 9.

In general, we are finding a curve that goes through one point (on the first line, $(1, 5)$) and is zero at the other two values of x (9 and 4). This follows the same breakdown as for the linear case.

First: how do we make the curve 0 at $x = 9$ and $x = 4$? This requires two killing terms:

$$L_1(x) \propto (x - 9)(x - 4) \tag{55}$$

Again, we can multiply $L_1(x)$ by anything (which we will do to satisfy other constraints), and it will still equal 0 at $x = 9$ and 4. Now, we need to make $L_1(x)$ equal 1 at $x = 1$, which we do by dividing by the killing terms at $x = 1$.

$$L_1(x) \propto \frac{(x - 9)(x - 4)}{(1 - 9)(1 - 4)} \tag{56}$$

Good: we now have a line that is 1 at $x = 1$, 0 at $x = 9$ and 4. For the last step, we want it to equal 5 at $x = 1$, so we just multiply by 5:

$$L_1(x) = 5\frac{(x - 9)(x - 4)}{(1 - 9)(1 - 4)} \tag{57}$$

All done! We can repeat this for $L_2(x)$ and $L_3(x)$, and (using linearity) we add them up to get:

$$y = L_1(x) + L_2(x) + L_3(x) \tag{58}$$

$$= 5\frac{(x - 9)(x - 4)}{(1 - 9)(1 - 4)} + 2\frac{(x - 1)(x - 4)}{(9 - 1)(9 - 4)} + 1\frac{(x - 1)(x - 9)}{(4 - 1)(4 - 9)} \tag{59}$$

$$= \frac{23}{120}x^2 - \frac{55}{24}x + \frac{71}{10} \tag{60}$$

Good. Let's generalize this. Consider a set of $n+1$ points (x_i, y_i). We can construct an n degree polynomial which goes through all $n + 1$ points by the following algorithm:

$f(x) = 0$
for $i = 1$ **to** $n + 1$ **do**
  $g(x) = 1$
  for $j = 1$ **to** $n + 1$, $j \neq i$ **do**
    $g(x) * = (x - x_j)/(x_i - x_j)$
  end
  $f(x) + = y_i \; g(x)$
end

Algorithm 1: Polynomial Interpolation

This algorithm can be used in two ways.

1. Symbolically constructing the interpolating polynomial: you can follow it step by step and write down the polynomial.
2. As a way to evaluate the polynomial at a specific point: you could write a `C` function that takes in two arrays for x_i and y_i, then a value x, and calculate the interpolating polynomial at that point.

This type of curve fitting is very useful for some applications (it's the basis of finite element methods, or FEM) but not for all. One large issue is the situation of data overfitting. A simple example presented later in class is fitting to measured data from projectile motion.

- Show this is bad for a large number of points.
- Motivate χ^2 fitting: derive average.
- Segway for mean, error, etc.
- General χ^2.

When one has a large number of data points, it becomes unreasonable to exactly fit a polynomial to the data. On one hand, the fits tend to look poor—an exact polynomial fit may oscillate wildly. On the other hand, it may not even be desirable: what if the data has error bars? It doesn't make sense to try to exactly fit the data in that case, but try to fit a curve through the error bars.

We're only going to scratch the surface of data fitting in this class. For now, we're going to motivate the most common type of fitting: least linear squares.

Consider a set of measurements y_i, $i = 1, 2, ...N$, optionally with standard errors σ_i (if there is no standard error, assume it is just 1), taken at locations (or times) x_i, respectively. You may have some reason to assume the data is well described by some polynomial function—a good example of this is projectile motion, which is described by parabolas, or quadratic polynomials. In general, a polynomial of degree m can be written:

$$f(x; a_0, ..., a_m) = a_0 + \sum_{i=1}^{m} a_i x^i, \tag{61}$$

where the a_i's (including a_0) are fit parameters.

Least squares fitting aims to find values for the a_i's which minimize the distance between the fit curve and the data points y_i.

Insert motivation here...

The fit coefficients a_i can be found by performing Gaussian elimination on the following augmented matrix:

$$\left(\begin{array}{ccccc|c} \sum_{i=1}^N \frac{1}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^2}{\sigma_i^2} & \cdots & \sum_{i=1}^N \frac{x_i^m}{\sigma_i^2} & \sum_{i=1}^N \frac{y_i}{\sigma_i^2} \\ \sum_{i=1}^N \frac{x_i}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^2}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^3}{\sigma_i^2} & \cdots & \sum_{i=1}^N \frac{x_i^{m+1}}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i y_i}{\sigma_i^2} \\ \sum_{i=1}^N \frac{x_i^2}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^3}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^4}{\sigma_i^2} & \cdots & \sum_{i=1}^N \frac{x_i^{m+2}}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^2 y_i}{\sigma_i^2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sum_{i=1}^N \frac{x_i}{\sigma_i^m} & \sum_{i=1}^N \frac{x_i^{m+1}}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^{m+2}}{\sigma_i^2} & \cdots & \sum_{i=1}^N \frac{x_i^{2m}}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^m y_i}{\sigma_i^2} \end{array}\right) \tag{62}$$

8 Lecture Notes: FFT as recursion for divide and conquer

We are going to consider computations using Fourier series. They are used to pick out frequencies in an oscillation signal and lots and lots of application in signal processing and transmission of information (telephones, radios, networks, etc.) The classic mechanical exmple is the simple pendulum. In small ocillation it has single harmonic motion used in grandfather clocks to keep time. There is a cute Mathematical Notebook on github to understand the single pedulum spectrum which he will show off in class. At large amplitude (no friction) the pendulum is anharmonic with a fixed period but a range of harmonic modes. Of course now very accurate clocks use solid state oscilators. See the book Longitude to see an amazing history of getting a clock accurate enough to keep British ship from crashing into the rocks as the land on a shore! (see https://en.wikipedia.org/wiki/Longitude_(book) First prize every provide for an Engineering advance.)

Such oscillators can be made of "springs", "pendulums", LRC circuits etc [3] Almost anything will oscillate around a stable point with **exactly** the same equation. Realistically need to add some damping or viscous drag to "F = ma":

$$m\frac{d^2x}{d^2t} = F_{spring} = -kx \quad \text{and damping} \quad \frac{d^2x}{dt^2} = F_{drag} = -\Gamma\frac{dx}{dt} \tag{63}$$

Recall the second equation if exponential $x = x_0e^{-\Gamma t}$ if try $x = x_0exp[\lambda t]$ so that $m\lambda^2 = -k - \lambda\Gamma$ and define $\omega_0^2 = k/m, \gamma = \Gamma/2m$ and solve quadratic equation! Therefore two solutions:

$$x = x_0e^{-\gamma t}\ e^{\pm i\sqrt{\omega_0^2 - \gamma^2}t} \tag{64}$$

For an LRC circuit $x \to Q, m \to L, k \to 1/C, \Gamma \to R, F \to V$. Non-linearities come in all sorts of forms! Pendulum $x \to \theta, m \to I, F \to gl\sin(\theta)$. Coupling in all sorts of forms. For more fun drive them with an force like $F_{ext} = V_{ext} = c_0\cos(\omega t + \phi_0)$ or a random noise !! See lots of (too much!) material on harmonic ocillator at https://en.wikipedia.org/wiki/Harmonic_oscillator Ciricuits simulator (analogue computers) and digital computer are essential!

8.1 Fast Fourier Transform

The fast Fourier transform is a classic algorithm very important to analyzing signals of all kinds. It was invented by Gauss (who else!) but credited to Cooley and Tukey who rediscovered without knowing Guass' earlier application: https://en.wikipedia.org/wiki/Fast_Fourier_transform

```
Gauss, Carl Friedrich,"Theoria interpolationis methodo nova tractata",
Werke, Band 3,265327 (Knigliche Gesellschaft der Wissenschaften, Gttingen, 1866)

Cooley, James W.; Tukey, John W. (1965). "An algorithm for the machine
calculation of complex Fourier series". Math. Comput. 19:
297301. doi:10.2307/2003354.
```

If a signal (sound, vibrations, circuits, etc.) are oscillatory in nature, it is better to replace a Polynomial fit with a fit to a sinusoidal expansion. Suppose, for example, you sample a signal at N "times" $y_n = f(t_kn)$ for $n = 0, 1, 2, \cdots N-1$ for times at equal intervals $t_n = n\Delta t$. The Fourier series for a discrete sample of a "time" sequence with N terms $n = 0, 1, \cdots N-1$ can be expanded in a series of sines and cosines:

$$y_n = a_0 + \sum_{k=1}^{N_0}[a_k\cos(2\pi kn/N) + b_k\sin(2\pi kn/N)] \tag{65}$$

[3] See LRC circuits:https://en.wikipedia.org/wiki/RLC_circuit

where $2N_0 + 1 \leq N$. Just as for a polynomial fit we can choose to have a perfect fit with $2N_0 + 1 \leq N$. Note thata this fit in fact assume it is good to approximate the signal can be assumed to be (approximately) periodic for $t \to t + N$ with frequences: $\omega_k = 2\pi k/N$. (Warning: In our FT formalism we will use "time" to "frequence" with $t_n \to \omega_k$ interchangeably with "space" to "wave number" with $x \to k$. Both of course use the same formulae.)

8.2 Complex Numbers and Discrete Fourier Series

New complex arithmetic. Special case

$$\cos(\theta) + i\sin(\theta) \equiv e^{i\theta} \tag{66}$$

Prove it by Taylor series! This is the easy way to get all sort of trig identities, e.g. $e^{i(\alpha+\beta)} = e^{i\alpha}e^{i\beta}$ is the sum of angle in complex form:

$$\cos(\alpha+\beta) + isin(\alpha+\beta) = (\cos\alpha\cos\beta - \sin\alpha\sin\beta) + i(\cos\alpha\sin\beta + \sin\alpha\cos\beta) \tag{67}$$

Representing sin's and cos's as complex phases:

$$\cos(\theta) = \frac{e^{i\theta} + e^{i\theta}}{2} \quad , \quad \sin(\theta) = \frac{e^{i\theta} - e^{i\theta}}{2} \tag{68}$$

The expansion real Fourier expansion in Eq.65 exactly equivalent (and more more more conveniently written) as

$$y_n = \sum_{k=0}^{N-1} c_k e^{2\pi ikn/N} \tag{69}$$

This tansforms N coefficients c_k into N vaules $y_n = f(t_n)$ so is a lineare equation ($y_n = A_{nk}c_k \to c_k = A^{-1}_{kn}y_n$ that can in principle be solved.This is a set of linear equation in the coeffients. We could solver for c_k by Gaussian elimination. This is a $O(N^3)$ approach. There is a much better trick. The inverse is in fact exactly

$$c_k = \frac{1}{N}\sum_{k=0}^{N-1} y_k e^{-i2\pi nk/N} \tag{70}$$

Why? It is one line to prove. Just substute one into the other and use

$$\sum_n e^{2\pi ikn/N} e^{-2\pi ik'n/N} = \delta_{kk'} \tag{71}$$

lets redefine $\omega_N^n \equiv e^{i2\pi n/N}$ the "N-th roots of unit".

We have seen one case of `devide and conquer` in searching a list using bisection. We start with an order list

```
int a[N];  // a  = {2, 13, 42, 45, 69, ...., 1101}
```

The search for a `key = 560` is naively on average $O(N)$ just scan the list checking

```
if(key == a[i])
   {
     cout << " i = " << i << "   " << a[i] << endl;
     return 0;
   }
```

We then compared with devide and conquer by bisection. The replace the number of step $T(N) = \text{const}N$ to a recursive count that relpaces $T(N)$ by a half size probem with $T(N/2)$ set. We can write an equation for this

$$T(N) = T(N/2) + c_0 \tag{72}$$

where c_0 the nuber of steps to devide it (2 or 3). The causes to estimate the steps to be at moast $n = \log_2(N)$ devisions becuase the $N \to N/2 \to N/4 \cdots \to 1$. We can show that this works in this equation:

$$c\log(N) = c\log(N/2) + c_0 \tag{73}$$

if choose $c = c_0/\log(2)$. We are now going to use this devide and conquer for a Fourrier expansion.

8.2.1 FFT details

Let's look at this carefully. The Fourier transform [4] (expansion) is

$$y_k \equiv \mathcal{FT}_N[c_n] = \sum_{n=0}^{N-1} (\omega_N^n)^k a_n = \sum_{n=0}^{N-1} e^{i2\pi nk/N} a_n \tag{74}$$

The trick is use a binary representation of : $n = n_0 + 2n_1 + 2^2 n_2 + ... + 2^p n_p$

$$\omega_N^n = \omega_N^{n_0}\omega_{N/2}^{n_1}\omega_{N/4}^{n_2}...\omega_2^{n_p} \tag{75}$$

$$\sum_n \omega_N^{nk} = \sum_{n_0=0,1} \omega_{N/2}^{n_1 k} \sum_{n_1=0,1} \omega_N^{n_0 k} ... \sum_{n_p=0,1} \omega_2^{n_p k} \tag{76}$$

If we do low bit first, we can split polynomial into even/odd pieces:

$$y_k = \sum_{n=0}^{N/2-1} e^{i2\pi nk/(N/2)} a_{2n} + \omega_N^k \sum_{n=0}^{N/2-1} e^{i2\pi nk/(N/2)} a_{2n+1} \tag{77}$$

Therefore, one N = two N/2 Fourier transforms.
low k:

$$y_k = \sum_{n=0}^{N/2-1} e^{i2\pi n/(N/2)} [a_{2n} + \omega_N^k a_{2n+1}] \tag{78}$$

high k:

$$y_{k+N/2} = \sum_{n=0}^{N/2-1} e^{i2\pi n/(N/2)} [a_{2n} - \omega_N^k a_{2n+1} \tag{79}$$

If we do high bit first, we can split polynomial into low/high pieces:

$$y_k = \sum_{n=0}^{N/2-1} e^{i2\pi nk/N} a_n + e^{i\pi k} \sum_{n=0}^{N/2-1} e^{i2\pi nk/N} a_{n+N/2} \tag{80}$$

Therefore, one N = two N/2 Fourier transforms.
even k:

$$y_{2\tilde{k}} = \sum_{n=0}^{N/2-1} e^{i2\pi n\tilde{k}/(N/2)} [a_n + a_{n+N/2}] \tag{81}$$

[4] Since the FT and its inverse is essential the same up to a sign and convention of where to put the $1/N$ or even nicer to use $1/sqrtN$ in both there is now "correct" convention. Here I have use a_n instead of c_n as well! Using library FFT codes, the user has to reconcile her use with the software! That's life. Don't fight it!

odd k:

$$y_{2\tilde{k}+1} = \sum_{n=0}^{N/2-1} e^{i2\pi n\tilde{k}/(N/2)}\omega_N^n[a_n - a_{n+N/2}] \tag{82}$$

This pattern as a data flow is called the butterfly images:
It is a recursive pattern high vs low bit order is going right to left or left to right! Thus,

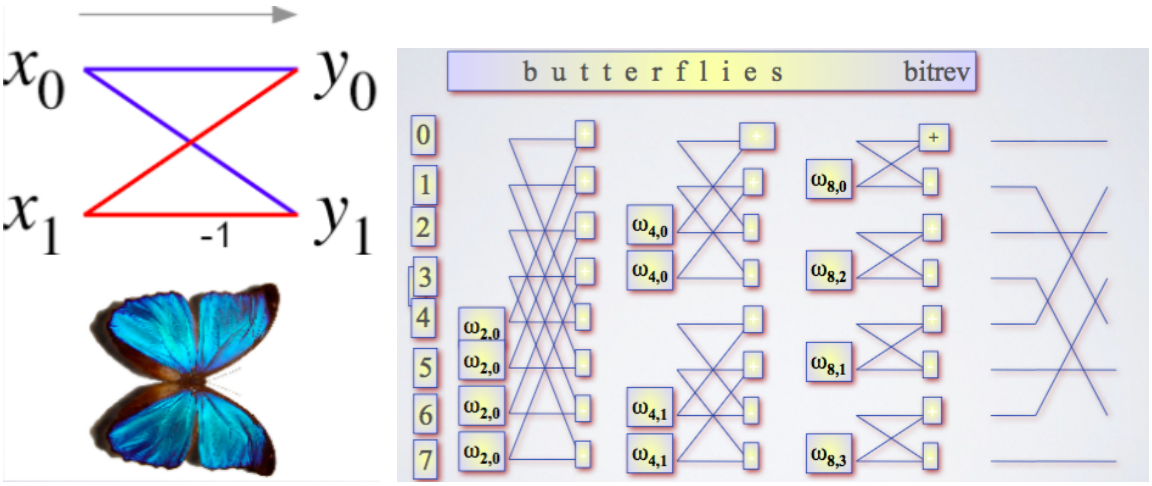


Figure 7: MAKE A CAPTION

$$T(N) = 2T(N/2) + c_0 N \Longrightarrow T(N) \sim N\log(N) \tag{83}$$

And in summary,

$$y_k = \mathcal{FT}_{N/2}[a_{2n} + \omega_N^k a_{2n+1}] \tag{84}$$
$$y_{k+N/2} = \mathcal{FT}_{N/2}[a_{2n} - \omega_N^k a_{2n+1}] \tag{85}$$

$$y_{2k} = \mathcal{FT}_{N/2}[a_n + a_{n+N/2}] \tag{86}$$
$$y_{2k+1} = \mathcal{FT}_{N/2}\omega_N^n[a_n - a_{n+N/2}] \tag{87}$$

Note however that the "bits are reversed" in the network flow.

8.3 Complex Fourier Test Code

Using complex varible in C code is a bit strange at times. To show how it work here is a test code for the (slow) Fourier Transform. In Problem Set #5 you will be asked to add FFT routines and apply them to solve heat conduction.

mainFFT.cpp

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <complex>
using namespace std;
#include "fft.h"

int main()
{
  int N = 16;
  Complex * omega = new Complex[N];
  Complex * F = new Complex[N];
  Complex * Fnew = new Complex[N];
  Complex * Ftilde  = new Complex[N];

  makePhase(omega,N);

  /* Test slow FT */
  for(int x = 0; x < N; x++){
    F[x] = 2* sin( 2.0*PI*x/(double) N) + 4*cos( 2.0*PI*3.0*x/(double) N);
    //    cout<<" x = "<< x << "  F =    " <<  F[x]  << endl;
  }

  FT(Ftilde, F,omega,N);

  for(int k = 0; k < N; k++)
    cout<<" k = "<< k << "  Ftilde =   " <<  Ftilde[k]  << endl;

  FTinv(Fnew,Ftilde, omega, N);

  for(int x = 0; x < N; x++)
    cout<<" x = "<< x << "  F =    " <<  F[x]  << " : " << Fnew[x] <<  endl;

  /*  Place to write and test a recursive FFT */
  //FFT(*Ftiled, F, *omega, N);
  //FFTinv(*Ftilde,F,*omega, N);

  return 0;
}
```

fft.cpp

```
void makePhase(Complex *omega, int N )
{
  for(int k = 0; k < N; k++)
    omega[k] = exp(2.0*PI*I*(double)k/(double) N);
}

void FT(Complex * Ftilde, Complex * F, Complex * omega, int N)
{
  for(int k = 0; k < N; k++)
    {
      Ftilde[k] = 0.0;
      for(int x = 0; x < N; x++)
        Ftilde[k] += pow(omega[k],x)*F[x];
    }
}

void FTinv(Complex * F, Complex * Ftilde, Complex * omega, int N)
{
  for(int x = 0; x < N; x++)
    {
      F[x] = 0.0;
      for(int k = 0; k < N; k++)
        F[x] +=pow(omega[k],-x)*Ftilde[k]/(double) N;
    }
}
```

fft.h

```
#ifndef FFT_H
#define FFT_H

#include <iostream>
#include <fstream>


#include <cmath>
#include <complex>
using namespace std;
#define PI 3.141592653589793
#define  I  Complex(0.0, 1.0)
typedef complex <double> Complex;

void makePhase(Complex *omega, int N);
void FT(Complex * Ftilde, Complex * F, Complex * omega, int N);
void FTinv(Complex * F, Complex * Ftilde, Complex * omega, int N);
void FFT(Complex * Ftilde, Complex * F, Complex * omega, int N);
void FFTinv(Complex * F, Complex * Ftilde, Complex * omega, int N);

#endif
```

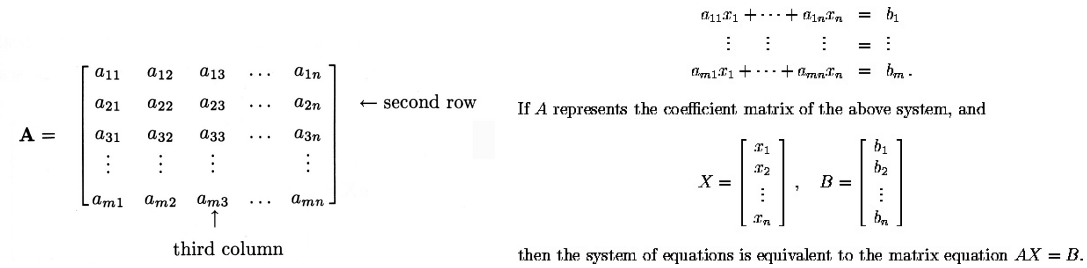
9 Lecture Notes: Introduction to Matrices

We have been using matrices in many of our method so far. In fact Matrix algebra is at the heart of almost all high performance computing in science and engineer. Finally tune libraries are develop and used to determine the "Top 500 List".

In computer science texts data structures for Lists/Graphs correspond exactly to Vctors/Matrices in numerical linear algebra.

```
Computer Science      <=====>   Linear Algebra
1D data structure     <=====>   Vectors
Graphs                <=====>    Matrices
```

Most algorithms clear correspondence as well.



$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\ \vdots \quad \vdots \quad\quad \vdots &= \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &= b_m . \end{aligned}$$

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Figure 8: On left .....On the right ....

The `Ax =b` linear algebra problem for square matrices we have already solve using Gaussian elimination. Mathematicians use the $A = ||a_{ij}||$ for matrixs and $X = [x_i], B = [b_i]$ for column vector (i.e 1xN matrices). Therefore we can write

$$\sum_j a_{ij}x_j = b_i \quad \Longleftrightarrow \quad AX = B \tag{88}$$

9.1 Matrices as Minimization of Qudratice Equations.

The linear algebra problem is closely related to quadratic polynomials (or "forms"). Let's start with a 1 verible problem. The quadratic problem is parabola

$$f(x) = \frac{1}{2}[(ax-b)^2 + c] = \frac{1}{2}[a^2x^2 - 2abx + b^2 + c] \tag{89}$$

Then mininum has zero derivative,

$$\frac{df(x)}{dx} = a(ax-b) = 0 \quad \text{or} \quad x = b/a \tag{90}$$

Now lets look at the general problem with N-variables (j = 1, ..,N)

$$S[x_1, ..., x_N] = \frac{1}{2}\sum_i (\sum_j a_{ij}x_j - b_i)^2 \tag{91}$$

The matrix $A = ||a_{ij}||$. What is the mininum of S?

$$\frac{\partial S}{\partial x_k} = \sum_i a_{ik}(\sum_j a_{ij}x_j - b_i) = 0 \tag{92}$$

Defining the transpose $A^T = ||a_{ji}||$ (switching rows and columns) this implies

$$a_{ik}(a_{ij}x_j - b_i) = 0 \Longleftrightarrow A^T(Ax - b) = 0 \tag{93}$$

(Using Einstein's trick: repeated indices i, j are summed. Always good to take advise from Einstein!). If we have enough N variable and N equations (e.g. A is square) then this can be solved by solving,

$$Ax = b \tag{94}$$

if we have too few variable we need to solve,

$$A^T Ax = A^T b \tag{95}$$

This is exact the problem we had with finding the χ^2 in Problem Set \$4. Let show this! Just need to be careful with notation,

$$\chi^2 = \sum_{i=0}^{N_0} \frac{(y_i - f(x_i))^2}{\sigma_i^2} = (y_i/\sigma_i - (x_i^n/\sigma_i)c_n) \tag{96}$$

so we can identify $b_i = y_i/\sigma_i$ and $a_{in} = (x_i^n/\sigma_i)$ to get the equation,

$$A^T A\, c = A^T b \Longleftrightarrow \sum_i (x_i^{m+n}/\sigma_i^2)c_n = \sum_i x_i^m y_i/\sigma_i^2 \tag{97}$$

So the "new" $A = ||a_{nm} = \sum_i (x_i^{m+n}/\sigma_i^2)||$ and the new $b_n = \sum_i \sum_j x_i^m y_i/\sigma_i^2$ **exactly** as we stated in Problem Set # 4. Solving $A^T Ax = A^T b$ is called the "pseudo-inverse". The best you can do the minimizatoin when you don't have enough variables!

9.2 Solviing the Matrix problem

We motivated the idea of smoothing images in class using a Mathematica notebook: the notebook has been posted to the github here: https://github.com/BU-EC-HPC-S16/EC500-High-Performance-Computing/tree/master/LaplaceFilter.

An $L \times L$ grayscale image can be represented as an $L \times L$ 2D array of numbers, representing the brightness of each pixel. In a standard greyscale image, each number may be an integer between 0 and 255 inclusive (8 bit grayscale images).

One useful operation on the image is *smoothing* or *blurring* the image. An intuitive way to do this is to replace each pixel value by some appropriate average of its neighbors. This is well represented by stencils. For example, replacing a pixel by an average of its neighbors can be written as:

```
px[x][y] = 0.25*(px[x+1][y] + px[x-1][y] + px[x][y+1] + px[x][y-1]);
```

where *px* is the 2D array.

As a brief remark: of course, pictures have boundaries, and we need to ask what happens at the boundary. We will discuss this more in depth later. A simple assumption is *periodic* boundary conditions: If we assume $x, y \in \{0, 1, \cdots, L-1\}$, we can assume $x + 1$ if $x = L - 1$ returns 0, and similarly $x - 1$ if $x = 0$ returns $L - 1$—wrapping around the grid. This can be accomplished in `C` by modifying the above code to use the modular arithmetic operator, `%`, to read:

```
px[x][y] = 0.25*(px[(x+1)%L][y] + px[(x-1+L)%L][y] + px[x][(y+1)%L] + px[x][(y-1+L)%L]);
```

where the weird convention for `-1` of `(x-1+L)%L` avoids a subtle issue of how the modular arithmetic operator handles the sign of input values.

In general, we don't just want to average, but do a weighted average of a site with its neighbors. The weighting parameter is `r`, where $0 <$`r`< 1. We can write the weighted average by:

```
px[x][y] = (1-r)*px[x][y]+r/4*(px[(x+1)%L][y] + px[(x-1+L)%L][y]
                               + px[x][(y+1)%L] + px[x][(y-1+L)%L]);
```

There are two limits to this method:

- `r = 0`: the pixel is replaced by itself, no change occurs.
- `r = 1`: the pixel is replaced by the average of its neighbors, the original example.

Some middle value of `r` is some appropriate combination.

This operation can also be written as a *stencil*. The stencil for the above combination can be written as:

$$\begin{pmatrix} 0 & r/4 & 0 \\ r/4 & 1-r & r/4 \\ 0 & r/4 & 0 \end{pmatrix} \tag{98}$$

To interpret the stencil, think of the center element as the location $(0, 0)$, which can be thought of as weighting the pixel itself: the piece `(1-r)*px[x][y]`. The rightmost $r/4$ corresponds to the location $(1, 0)$, which can be thought of as the contribution of the pixel to the right: `r/4*(px[(x+1)%L][y]`. The rest of the elements contribute analagously—the weighted average above can be completely encoded by this stencil (up to the assumption of periodic boundary conditions).

This is actually how the Mathematica notebook works: it defines the matrix given above `smoothMatrix` with a free parameter `rho` (the same as `r` we use in this document), and then it applies the function `ImageConvolve` to apply the stencil once. [5]

Back to the smoothing operation: let's get rid of worrying about boundary conditions (let's just assume we're sufficiently far from the boundary), and think about if there's some continuum physics at hand here.

Let's consider `px` as a function f, which we'll give some possible physical interpretations in a bit. The operation becomes:

$$f(x, y, t+1) = (1-r)f(x, y, t) + \frac{r}{4}\left[f(x+1, y, t) + f(x-1, y, t) + f(x, y+1, t) + f(x, y-1, t)\right] \tag{99}$$

You'll notice I added a piece t to the function: it can be thought of as the step iteration; it can also be thought of as *time*. We had this notation ± 1 because we were thinking of sites on a grid; let's give it some physical distance instead, which we'll call a. Similarly, we'll replace the $+1$ in time with $+\delta t$, for a little time

[5] There is a nice connection between applying stencils and Fourier transforms, where an operation called a `convolution` is applied: thus the word "Convolve" in the function name.

step. If this spacing between sites is small, we can perform a Taylor series up to order a^2. Let's see what we get:

$$\begin{aligned}
f(x,y,t+\delta t) &= (1-r)f(x,y,t) + \frac{r}{4}\left[f(x+a,y,t) + f(x-a,y,t) + f(x,y+a,t) + f(x,y-a,t)\right] \\
&\approx (1-r)f(x,y,t) + \frac{r}{4}\Bigg[f(x,y,t) + a\frac{df}{dx}(x,y,t) + \frac{a^2}{2}\frac{d^2f}{dx^2}(x,y,t) \\
&\qquad + f(x,y,t) - a\frac{df}{dx}(x,y,t) + \frac{a^2}{2}\frac{d^2f}{dx^2}(x,y,t) \\
&\qquad + f(x,y,t) - a\frac{df}{dy}(x,y,t) + \frac{a^2}{2}\frac{d^2f}{dy^2}(x,y,t) \\
&\qquad + f(x,y,t) - a\frac{df}{dy}(x,y,t) + \frac{a^2}{2}\frac{d^2f}{dy^2}(x,y,t)\Bigg] \\
&= (1-r)f(x,y,t) + \frac{r}{4}\left[4f(x,y,t) + 2\left(\frac{a^2}{2}\frac{d^2f}{dx^2}(x,y,t) + \frac{a^2}{2}\frac{d^2f}{dy^2}(x,y,t)\right)\right] \\
&= f(x,y,t) + \frac{ra^2}{4}\nabla^2 f(x,y,t),
\end{aligned}$$

where on the last line I implicitly defined

$$\nabla^2 = \frac{d^2}{dx^2} + \frac{d^2}{dy^2} \tag{100}$$

as the *Laplacian*. The Laplacian is denoted in some other texts as Δ. It's interesting to note that all dependence on the parameter r has disappeared in front of $f(x,y,t)$—it turns out this is related to a special property of this smoothing parameter, that it *preserves the average* of the values of all the pixels on each step.

We can also briefly do a series in δt. This gives us:

$$\begin{aligned}
f(x,y,t) + \delta t\frac{df}{dt}(x,y,t) &= f(x,y,t) + \frac{ra^2}{4}\nabla^2 f(x,y,t) \\
\frac{df}{dt}(x,y,t) &= \frac{ra^2}{4\delta t}\nabla^2 f(x,y,t)
\end{aligned}$$

This has a very special interpretation in physics and engineering as the *heat* or, more generally, the *diffusion* equation. In the former case, f can be more appropriately labeled as T, the temperature.

We can gain some good intuition for what our averaging iterations do from this physical interpretation. Let's say it's a hot, dry day, and we want to cool the kitchen down. One (really bad) way to do this is to open the fridge and leave it open. If it's 36 degrees in the fridge, and 80 in the room overall, over time the temperatures will even or average out, and the room will overall get cooler (negligibly, unless your fridge is huge—like a quarter of the room huge). This is the diffusion or heat equation in action, and in an "ideal" room (like a spherical cow) the average temperature will be preserved, just like how our smoothing operation preserved the average pixel intensity—cool!

We've seen that our averaging procedure corresponds to the difussion equation in the continuum limit. We can run this argument backwards, too—we can approximately solve diffusion problems (like temperature propagation) by using this discrete averaging procedure.

We'll use this example to motivate a few linear solvers.

Let's consider the case where we have a well insulated room. In one corner, we have the refrigerator at 36 degrees. The rest of the room is at 80 degrees. If we split our room up into an 8x8 square grid, we might initialize an array `T` as (assuming the fridge takes up 1/16 of the room).

```
for (int x=0; x<8; x++)
{
  for (int y=0; y<8; y++)
  {
    if (x < 2 && y < 2)
    {
      T[x][y] = 36.0;
    }
    else
    {
      T[x][y] = 80.0;
    }
  }
}
```

One thing to be careful about is how to scale to the continuum limit: as we refine the lattice more and more, a fixed fraction of the room should be the fridge. If we hardcode `L` as the length of the room, we can write it as:

```
for (int x=0; x<L; x++)
{
  for (int y=0; y<L; y++)
  {
    if (x < L/4 && y < L/4)
    {
      T[x][y] = 36.0;
    }
    else
    {
      T[x][y] = 80.0;
    }
  }
}
```

where we've now fixed the fridge to be in a set portion of the room.

Good! Now, we know that if we keep iterating with our averaging routine above, we will eventually *relax* the room to the average temperature. Let's assume at first we had periodic boundary conditions, so we can use the iterations we wrote above, hotswapping `px` for `T`. We will also store the updated results into a new array, `Tnew`, and copy it over when we're done.

```
done = 0;
while (done == 0)
{
  for (x = 0; x < L; x++)
  {
    for (y = 0; y < L; y++)
    {
      Tnew[x][y] = (1-r)*T[x][y]+r/4*(T[(x+1)%L][y] + T[(x-1+L)%L][y]
                                      + T[x][(y+1)%L] + T[x][(y-1+L)%L]);
```

```
    }
  }
  for (x=0;x<L;x++) { for (y=0;y<L;y++) {
    T[x][y] = Tnew[x][y];
  } }
  // check if we're done? How do we do that?
}
```

This is roughly how a set of iterations would work—of course, there's one important question. When do we stop iterating?

A good measure of when to stop is to look at the relative residual—how much is `T` changing on each iteration? We can measure this by looking at the residual (defined as a floating point `res`) between steps, for example:

```
done = 0;
while (done == 0)
{
  for (x = 0; x < L; x++)
  {
    for (y = 0; y < L; y++)
    {
      Tnew[x][y] = (1-r)*T[x][y]+r/4*(T[(x+1)%L][y] + T[(x-1+L)%L][y]
                                    + T[x][(y+1)%L] + T[x][(y-1+L)%L]);
    }
  }
  res = 0.0;
  for (x=0;x<L;x++) { for (y=0;y<L;y++) {
    res += (T[x][y]-Tnew[x][y])*(T[x][y]-Tnew[x][y]);
    T[x][y] = Tnew[x][y];
  } }
  res = sqrt(res);
  if (res < 1e-6)
  {
    done = 1;
  }
}
```

We notice that if we think of `T` and `Tnew` as vectors (in the geometric sense), `res` looks like the magnitude of the difference of the two vectors—this is a nice interpretation. Our quitting condition is that the magnitude of this vector, which quantifies how much the solution changed on each step, is (sort of arbitrarily) below `1e-6`, or 10^{-6}.

This way of checking the residual will be consistent throughout our discussion—our way of updating `T` will not be!

The method we've written above is known as *Jacobi iterations*. We update every single site with the old values, then copy all of the old values over. It's efficient, it can be easily parallelized, but it's not optimally fast. How do we speed it up?

One way to speed up the algorithm is to put updated values in as we compute them. This means, on each iteration, some elements `Tnew` are computed with some components of `T`, some components of "`Tnew`". To

keep comparing a residual, we need to tweak slightly how we save the pre-iteration values. This new way of updating the array is called *Gauss-Seidel* (of course it's Gauss), and it can be implemented like this:

```
done = 0;
while (done == 0)
{
  // Back up old values
  for (x=0;x<L;x++) { for (y=0;y<L;y++) {
    Told[x][y] = T[x][y];
  } }
  // compute the residual as we update
  res = 0.0;
  for (x = 0; x < L; x++)
  {
    for (y = 0; y < L; y++)
    {
      // update in place
      T[x][y] = (1-r)*T[x][y]+r/4*(T[(x+1)%L][y] + T[(x-1+L)%L][y]
                                   + T[x][(y+1)%L] + T[x][(y-1+L)%L]);
      res += (T[x][y]-Told[x][y])*(T[x][y]-Told[x][y]);
    }
  }
  res = sqrt(res);
  if (res < 1e-6)
  {
    done = 1;
  }
}
```

In practice (this can also be proven), *Gauss-Seidel* will converge to a fixed residual in half the number of residuals of Jacobi iterations. This came at a great cost, though, in the context of High Performance Computing! We cannot parallelize Gauss-Seidel because of data dependence—some values of `T` have already been updated, some haven't, at any given step `x` and `y`. So while Gauss-Seidel may be faster on a single core machine, that's where the benefits stop: Jacobi will roughly tie it on a two core machine (if the code is parallelized), and beat it from there.

Thankfully, there is a way to beat this—a "best of both worlds" type deal. To understand it, we need to discuss a bit of graph theory, and to do that, it actually helps to bring back the stencil we wrote down earlier:

$$\begin{pmatrix} 0 & r/4 & 0 \\ r/4 & 1-r & r/4 \\ 0 & r/4 & 0 \end{pmatrix}$$

On each application of the stencil, what sites get accessed, and what sites get updated?

- Only the central element gets updated on each iteration: we defined our stencil this way.
- Only the central and neighboring elements are accessed on each iteration. This stencil has a maximum range: *one site away*.

This access pattern implies that our lattice of points is an *bipartite* lattice with respect to the stencil: updating any one site only depends on the value of that site and its neighbors. Further, if a site is so-called *even*: `(x+y)%2==0`, it only depends on so-called *odd*: `(x+y)%2==1` neighbors, and vice versa. This lets us consider a fancier update algorithm, *even-odd* updates. On a single iteration:

- Update all *even* sites. Since this only depends on the site itself and its neighbors, which are all *odd*, this can happen out of order: there are no data dependency issues.
- Update all *odd* sites, which can be done in any order for the same reason.

In practice, this is just as fast as Gauss-Seidel, so we get that factor of two speed up compared to Jacobi. At the same time, though, there's no data dependence within the loops over all even or all odd sites, so we can parallelize these loops, just like Jacobi: we can get a parallelization speed up too.

A (sloppy) red-black implementation would look like this:

```
done = 0;
while (done == 0)
{
  // Back up old values
  for (x=0;x<L;x++) { for (y=0;y<L;y++) {
    Told[x][y] = T[x][y];
  } }
  // compute the residual as we update
  res = 0.0;
  for (x = 0; x < L; x++)
  {
    for (y = 0; y < L; y++)
    {
      // update in place all even sites
      if ((x+y)%2 == 0)
     {
        T[x][y] = (1-r)*T[x][y]+r/4*(T[(x+1)%L][y] + T[(x-1+L)%L][y]
                                     + T[x][(y+1)%L] + T[x][(y-1+L)%L]);
        res += (T[x][y]-Told[x][y])*(T[x][y]-Told[x][y]);
      }
    }
  }
  for (x = 0; x < L; x++)
  {
    for (y = 0; y < L; y++)
    {
      // update in place all odd sites
      if ((x+y)%2 == 1)
     {
        T[x][y] = (1-r)*T[x][y]+r/4*(T[(x+1)%L][y] + T[(x-1+L)%L][y]
                                     + T[x][(y+1)%L] + T[x][(y-1+L)%L]);
        res += (T[x][y]-Told[x][y])*(T[x][y]-Told[x][y]);
      }
    }
  }
  res = sqrt(res);
  if (res < 1e-6)
  {
    done = 1;
  }
}
```

There are fancy ways to further modify this—special memory layouts, blocking, etc—to make this even more

efficient, but in terms of writing a code that can be parallelized, this gets the point across. Of course, to get real speed up, we need better algorithms like multigrid, which we will get to later.

10 Lecture Notes: Multigrid Recursion to Speed up Matric Solvers

Multigrid is a recursive approach to speeding up the `Ax = b` matrix solver. It shares ideas of "devide and conquer" like bisection searching or merge sorting AND ideas for Fourrier mode decomposition! Really neat and useful! Brigg's "Multigrid Tutorial is an excellent source. See some really nice tutorial slides (free) and book (to purchase) at https://computation.llnl.gov/casc/people/henson/mgtut/welcome.html These slides have much much too many details for our needs so just look at the first 25 pages of slides. Here is a simpler outline of the idea.

10.1 1 D Laplacian Multigrid

Consider first the 1D problem on a line from $x = 0$ to $x = h(L-1)$. **To make life easy we will take a periodic problem with $L = 2^n$ as power of two and let $x = 0$ and $x = hL$ be the same point.** We need to solve [6]:

$$-\frac{d^2\phi(x)}{d^2x} \simeq \frac{2\phi[x] - \phi[x+h] - \phi[x-h]}{h^2} + m^2\phi[x] = b[x] \tag{101}$$

For the multigrid formalism it is crurial keep track of the lattice spacing h. To define the basic `Ax=b`' problem all we need to know is the how to apply the matrix A a vector: `vecOut=A*vecIN` In code this can be provide by a simple function, such as `void AmatVec(double *vecOut, double *vecIn, int N)`.

So the equation,

$$\phi_{out}[x] = A_h\phi[x] = \frac{2\phi[x] - \phi[x+h] - \phi[x-h]}{h^2} + m^2\phi[x] \tag{102}$$

is equivalent to

$$A_h\phi = \frac{1}{h^2}\begin{bmatrix} 2+h^2m^2 & -1 & 0 & 0 & \cdots & 0 & -1 \\ -1 & 2+h^2m^2 & -1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 2+h^2m^2 & -1 & \cdots & 0 & 0 \\ 0 & 0 & -1 & 2+h^2m^2 & \cdots & 0 & 0 \\ \cdots & & & & & & \\ 0 & 0 & 0 & 0 & \cdots & 2+h^2m^2 & -1 \\ -1 & 0 & 0 & 0 & \cdots & -1 & 2+h^2m^2 \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \cdots \\ \phi_{L-2} \\ \phi_{L-1} \end{bmatrix} \tag{103}$$

But it is much more compact and does not require a an $L \times L$ array (which is terrible data structure for a sparse matrix since most elements are 0). In higher dimension$D > 1$, it is avery awakard to even illustrate the sparce Laplace matrix. **In the jargon of HPC this is a *matrix-free* approach where A is a linear *stencil* operators on *structured grid* so the iterative solver just need to have the *MatVec* defined.** Generalization to unstructured grids are also done this way treating the grid as a genearl graph.

We could take $h = 1$ but let's leave it in for now becuase the idea of Multigrid is consider a sequence problems (like in the FFT) with $h \to 2h \to 2^2h \cdots$ and lattice points $L \to L/2 \to L/2^2 \cdots$. Suppose after iterating a little you have an approximation $\phi[x]$ to the exact solution $\phi_{exact}[x]$

$$\phi_{exact}[x] = \phi[x] + e[x] \simeq \phi[x] \ . \tag{104}$$

There is still a small"residual" mismatch in our equation above

$$r[x] = b - A_{xy}\phi[y] \neq 0 \tag{105}$$

The error obeys the same matrix eqution

$$A_h e = r \tag{106}$$

[6] We also slipped in a little value for $m^2 > 0$ to stabelize this equation. Without this the solution is ambiguous becuase if $\phi[x]$ is a solultion so is $\phi[x] + \text{const}$.

with the residue as the RHS. Zero residue implies zero error. Ths follows trivially from $A(\phi_{exact} - \phi) = b - A\phi = r$. The multigrid algorithm use this error equation on all coarse lattices. For eample a two level multgrid proceedure is the following:

1. The problem, approximated on fine lattice with lattice spacing h and $x = 0, 1, 2, \cdots L-1$ sites, has error satisfying:
$$A_h e = r \tag{107}$$

2. We average (or project) residue on each block (even and odd pairs in 1D) to coarse sites $\hat{x} = 0, 1, 2, \cdots L/2-1$.
$$\hat{r}[\hat{x}] = \frac{r[2\hat{x}] + r[2\hat{x}+1]}{2} \tag{108}$$

3. Find the error on the $2h$ grid using the coarse approximation,
$$\hat{A}\hat{e}[\hat{x}] = \hat{r}[\hat{x}] \ . \tag{109}$$

4. Correct the fine solution by setting $\phi[x] = \phi[x] + e[x]$ by copying back (or interpolate) the coarse error to the fine lattice.
$$e[2\hat{x}] = \hat{e}[\hat{x}] \quad , \quad e[2\hat{x}+1] = \hat{e}[\hat{x}] \tag{110}$$

Now we have guess the coarse operator, $\hat{A}$? The obvious choice is to just look at Eq. , replace replacement: $h \to 2$
$$\hat{A}\phi = A_{2h}\phi = \frac{2\phi[x] - \phi[x+2h] - \phi[x-2h]}{4h^2} + m^2\phi[x] \tag{111}$$
There is another popular form called the **Gelerken Prescripition:** Instead we use the non-square matrices for projection P and interpolations Q defined by there action above as
$$\hat{r}[\hat{x}] = \sum_x P_{\hat{x},x} r[x] = \frac{r[2\hat{x}] + r[\hat{x}+1]}{2} \tag{112}$$
and
$$e[x] = \sum_{hatx} Q_{x,\hat{x}} \hat{e}[\hat{x}] = \hat{e}[x/2] \tag{113}$$
respectively. The idea is to approximate the fine operator equation by
$$Ae = r \quad \to \quad (PAQ)Pe = Pr \tag{114}$$
as we take $h \to 2h$. The resulting Gerlerkin form defines $\hat{A} = PAQ$ (or $\hat{A}\hat{e} = \hat{r}$). A little tedeous algebra gives,
$$\hat{A}\hat{e}(\hat{x}) = \frac{2\phi[x] - \phi[x+2h] - \phi[x-2h]}{2h} + m^2\phi[x] = b[x] \tag{115}$$
Strange! Only one factor of h! in the first term! Not the naive scaling.

We note in our code `mg.c` maniputed the quation into standard form
$$\hat{e}[\hat{x}] = \alpha(lev)(\hat{e}[\hat{x}+1] + \hat{e}[\hat{x}-1]) + \hat{r}[\hat{x}] \tag{116}$$
on the coarse lattices at levele level $lev = 1, 2, \cdots$ ... with spacing $h \to 2^{lev}$. So the scaling form uses $\alpha(lev) = 1/(2 + 2^{2lev}m^2)$ or $\alpha(lev) = 1/(4 + 2^{2lev}m^2)$ in 2D where as the Gelerkin form uses $\alpha(lev) = 1/(2 + 2^{lev}m^2)$ or $\alpha(lev) = 1/(4 + 2^{lev}m^2)$ in 2D. Both work very well. (At the top level since we set h = 1 is not changed.) In fact this may be a little better as way to do Multgrid. Magic

10.2 Fixed Boundary Conditions

In some ways this is easier. We can drop the m^2 term and consider an expanded interval $x = -1, 0, 1, ..., L-1, L$ including fixed boundaries values $\phi[0] = T[-1] = T_a$ and $T[L] = T_b$. (I have use T because maybe the more physically appealing case is a temperature field as in our Project.) Now the variable we adjust are only the L interior points $\phi[x] \to T[x]$ for $x = 0, 1, 2, \cdots, L-1$ The matrix is $L \times L$ acting on these points

$$AT[x] = \frac{T[x+1] + T[x-1] - 2T[x]}{h^2} + b_0[x] \quad \text{for} \quad x = 0, 1, 2, \cdots, L-1 \tag{117}$$

We need to be careful in using this in Multigrid. The $L \times L$ matrix matrix that acts on the variable that you update is really:

$$AT = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 & 0 \\ 0 & 0 & -1 & 2 & \cdots & 0 & 0 \\ \cdots & & & & & & \\ 0 & 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & 0 & \cdots & -1 & 2 \end{bmatrix} \begin{bmatrix} T[0] \\ T[1] \\ T[2] \\ T[3] \\ \cdots \\ T[L-2] \\ T[L-1] \end{bmatrix} \tag{118}$$

You drop the term in the corners at $0, 0$ and $L-1, L-1$. When we have zero temperature boundaries, this all you need to do. But in general the fixed temperatures really contributes to the source. So the correct source for the $Ax = b$ problem is

$$b[x] = b_0[x] + \delta_{x,0} T_a/h^2 + \delta_{x,L-1} T_b/h^2 \tag{119}$$

include two contributions form the boundaryies.

`The lesson here is the Boundaries are really sources which makes a lot of sense physically.` You can input heat in the middle with $b_0[x]$ or on edges with $b[x]$. Now the boundaries just go along for the ride when we define the coarse level in multigrid:

$$r[x] = b[x] - AT[x] \quad \text{for} \quad x = 0, 1, 2, \cdots, L-1 \tag{120}$$

Just remember to keep the extra terms in $b[x]$ and drop the term in A as if the boundary was zero! Doing this in 2D is the same except on you have $T[x][y]$ for $x, y = 0, 1, 2, \cdots, L-1$ in the interior and lines on the sides of the square: boudaries are lines

$$h^2 b[x][y] = h^2 b_0[x][y] + \delta_{x,0} T[-1][y] + \delta_{x,L-1} T[L][y] + \delta_{y,0} T[x][-1] + \delta_{y,L-1} T[x][L]] \tag{121}$$

Of course you have 2 by 2 blocks to proeject and interpolate. For example,

$$\hat{r}[\hat{x}][\hat{y}] = Pr[\hat{x}][\hat{y}] \frac{1}{4} (r[2\hat{x}][2\hat{y}] + r[2\hat{x}+1][2\hat{y}] + r[2\hat{x}][2\hat{y}+1] + r[2\hat{x}+1][2\hat{y}+1]) \tag{122}$$

`This is basically same as the briliant idea from class BUT now when you put the boundary sorces into the` $b[x][y]$ `everything happend automatically as you go to lower levels by passing in the residue and you never have to worry about it! Pretty slick!`

Comment on coding: Just like in the case of *ghost zones* for the MPI it is convenient in C to layout the arrays starting form zero with two extra slots for the boundaries. For example `double T[L+2][L+2]` In the MPI code uses a cute trick with and off-set is to use an off-set index $T[x+1][y+1]$ so with interior values are given by $x, y = 0, 1, \cdots L-1$ the boundaries at $x, y = -1$ and $x, y = L$. Defining off-set array, $T_c[x][y] = T[x+1][y+1]$, the boundaries are now stored in $T_c[-1][y], T_c[x][-1]$ and $T_c[L][y], T_c[x][L]$ as they should. This allows one to have a very simple iterative scheme for the interior such as

$$T_c[x][y] = \frac{1}{4} (T_c[x+1][y] + T_c[x-1][y] + T_c[x][y+1] + T_c[x][y-1]) + b_0[x][y] \tag{123}$$

with loops over $x, y = 0, 1, \cdots, L-1$. The boundaries are automatically included as if they were in $b_0[x][y]$!. (**Note more jargon: Boundaries shared between MPI processes are called *ghost zones***

11 2017 Project Schedule & Topic

Schedule:

- Monday April 3 and 5: Set up teams of 2 each on Heat Dissapation.
- Wednesday April 12: Present Team Project Plan: Who is responsible for what part. 4 slides.
- Wednesday April 24: Code and Written report: A separate one page on individual contribution and team assessment.
- Monday May 1 and May 3 : Final Project Presentation via Slides: One hour ecah team.

Project Requirements.

- Each team must write their own code and go in different directions. However sharing ideas during the Lectures is ok – in fact encourged.
- Problem statement, tasks and goals with a 4 -5 slide proposal.
- Statement of the equations to solve and the basic algorithm.
- An assesment of algorithm efficiency and error of results.
- An example of parallelization by either openMP or MPI
- A graphics presentation of result.
- Final project should be written up in a short report and set of slides.

Project Topic:

The general topic is the use of 2D Linear solvers and paralleliztion. The engineering problem is to understand temperture distribution on a 2D chip. The basic heat equation is described in detail in:

https://en.wikipedia.org/wiki/Heat_equation

The basic problem can be static (no time dependence) but the simple solver (local) iterative solvoer does give you the time dependence for free! If you are ambitious you many extend the input heat to random fluctuatation. You might even consider at magentic system replacing temperature to see how it effects magnetization. But start simple and see how far you can go.

Here is start for reference: https://en.wikipedia.org/wiki/Thermal_simulations_for_integrated_circuits We all can collaborate in googling for useful information.

The topic is flexible so you should do research on this topic and pick a realistic goal given what the course has taught and teams talents. **In the remaining meetings there will be a lot of hands on help in the Lab to suggest solutions.** Part of the report can be library research on more advance methods that you might pursue if you were to take on this professionally!

11.1 Some ideas and References

Part of the project is for each team to go off and find neat ideas to explore. First let us consider the basic time dependent equation described in https://en.wikipedia.org/wiki/Heat_equation

We are looking at this in 2D. So let think about it. Heat get "pumped" in at rate of $\dot{q}(x, y, t)$ per unit area at location x, y. It has a flux of $k(x, y)\nabla T$ where is k the conductivity for heat to flow from high temperture to low and its divergence $(\nabla \cdot k\nabla T(x, y, t))$ tells how much total heat is conducted away from that point at x, y. Finally what heat remains raise the temperture $\rho c_p \dot{T}(x, y, t$ depending on the specific heat times the density of the material $\rho(x, y)c_p(x, y)$. All this result in this equation!

$$\dot{q} = \rho c_p \frac{\partial T(x, y, t)}{\partial t} - \nabla \cdot (k\nabla T(x, y, t)) \tag{124}$$

If we ignore the x, y dependence of k, ρ, c_p this is just the equation we are using to solver the `Ax = b` problem in descrete form!

$$\frac{T[x, y, t + \delta t] - T[x, y, t]}{\delta t} = k\frac{T[x + h, y, t] + T[x - h, y, t] + T[x, y + h, t] + T[x, y - h, t] - 4T[x, y, t]}{h^2} + \dot{q}[x, y, t] \tag{125}$$

where for simplicity we have redefined $k/\rho c_p \to k$ and $\dot{q}/\rho c_p \to \dot{q}$. Next multiplying through by δt and defining $\alpha = 4k\delta t/h^2$ we get

$$T[x, y, t+\delta t] = (1-4k\delta t/h^2)T[x, y, t]+(k\delta t/h^2)(T[x+h, y, t]+T[x+h, y.t]+T[x+h, y, t]+T[x+h, y, t])+\delta t\dot{q} \tag{126}$$

This is just the Jacobi iteration with an overrelaxation perameter:$\alpha = 4k\delta t/h^2$ and

$$T[x, y, t+1] = (1-\alpha)T[x, y, t]+\alpha(T[x+h, y, t]+T[x-h, y, t]+T[x, y+h, t]+T[x, y-h, t])/4+\alpha b(x, y) \tag{127}$$

Small α is small time steps (very under-relaxed). The standard solver used $\alpha = 1$. Ok now we see that we can do all sorts of real heat conduction problems.

Putting x, y dependence back in gives a grid that appears to have uneven spacing. This means that some parts are made of different materials and heat may flow preferentialy in different directions. This is not so hard to do. It just puts different weigths ($w_i(x, y)$) on the 4 link from x, y to the 4 neighbors $i = E, N, W, S$. (To preserve the constant temperature the local weight of the $4T$ term is $w_E(x, y) + w_N(x, y) + w_W(x, y) + w_S(x, y)$.) If some one is interested, we can try that too. This is an iteresting but subtle problem of heat conduction in non-homogeneous anisotropic media. Finally there are boundaries that we can hold to fixed values $T(t)$ or even fixed flux. Do some research in chip heat condition and algorithmic issues for simulating the Heat equation.

So start with a very simple iterations on a square lattice and get some graphics going to see how it evolves. The project can consider the physics of the evolution or the algorithmics of the evolution as you wish.

COMMENT: Our multigrid method solves a time independent problem, i.e. setting

$$\partial T(x,y,t)/\partial t = 0 \tag{128}$$

Eq. 124 above) . Taking $k = 1$ and add a "mass" term to stabelize it on a periodic lattice this equation is

$$b = m^2 T(x,y) - \nabla^2 T(x,y) \tag{129}$$

instead of Eq. 124 . The same basic steps now led to the matrix problem,

$$0 = -\frac{T[x+h,y] + T[x-h,y] + T[x,y+h] + T[x,y-h] - 4T[x,y]}{h^2} + m^2 T[x][y] + b[x][y] \tag{130}$$

To transfer to coarser lattice changing $h \to 2h \to 4h \cdots$ it is important see the spacing h explicity:

$$(4 + h^2 m^2) T[x][y] = T[x+h,y] + T[x-h,y] + T[x,y+h] + T[x,y-h] + b[x,y] \tag{131}$$

This can be rescaled to put it in the standard form:

$$T[x,y] = \text{scale}_h (T[x+h,y] + T[x-h,y] + T[x,y+h] + T[x,y-h]) + \text{scale}_h b[x,y] \tag{132}$$

where

$$\text{scale}_h = 1/(4 + h^2 m^2) \tag{133}$$

1D Multigrid problem; Now let us look at the 1D example in the notation of the example code `mg.c` so it is easy to modify it. In 1D the scaling factor is

$$\text{scale}(lev) = 1/(2 + 2^{2lev} h^2 m^2) \tag{134}$$

for levels: $lev = 0, 1, cdots$. (Of course in the code we can now set $h = 1$.)

WARNING: Cute trick in code: We use only two arrays $\phi[x]$ and $res[x]$ on all levels, letting $\phi[x]$ play the role of both error and phi! How do we do this?

On lev = 0, we iterate

$$\phi[x] = \text{scale}(0) * (\phi[x+1] + \phi[x-1]) + res[x] \tag{135}$$

starting with $\phi[x] = 0$ so that $\text{scale}(0) b[x] = \text{scale}(0) b[x] - A\phi[x] = res[x]$ for $\phi = 0$.

Going down ('fine to coarse') we project using

$$res[x] = \text{ Project}(res[x]) \tag{136}$$

and iterate:

$$\phi[x] = \text{scale}(lev) * (\phi[x+1] + \phi[x-1]) + res[x] \tag{137}$$

Going back up ("coarse to fine") we combined "interpolate plus add" routines,

$$\phi_{fine}[x] = \phi_{fine}[x] + \text{Interpolate}(phi_{coarse}[x]) \tag{138}$$

Here `phi_{coarse}[x]` is the error comtributing to the solution `phi_{fine}[x]`. Cute yes!

Of course if you wish you could introduce two arrays $\phi[x]$ and $err[x]$ at each level and split "interpolate plus add" into two routines by first intropolate and then add. This waste space and function calls but it is easier to understand. **This is just one of many examples of the dangers of writing optimized code that is harder to read!**

Ok so the project can choose to two directions (or a little of both): (1) Solver algorityms for the Laplace eqution or (2) Time dependence of the Heat equartion. Test of how fast this does it and how to put in MPI is sufficient.

Both can put in various sources of "heat" or boundary condition of heat flow or even variation in conductivity.

BUT remember the basic engineering moto: **KISS**

12 Lecture Notes: Conclusions/Fordward

There are of course a huge number or topics on which we have barely glimpsed. So here is a **concluding forward look**. Believe it or not what we have done here is an entre into the most useful and common feature of all numerical methods. A few ideas and millions of applications!

Our project has been to solver the Laplace equation $\nabla^2\phi = \dot{q}$ or convert to it as the end point of the thermal diffusion $\dot{T} = -\nabla^2 T - \dot{q} \to 0$. These are linear equation of the elliptical $(x)$and parabolic (x) form. There is one more imporant example the hypebolic or wave equation,

$$\frac{\partial^2\phi}{\partial t^2} = \nabla^2\phi + f(x,t) \tag{139}$$

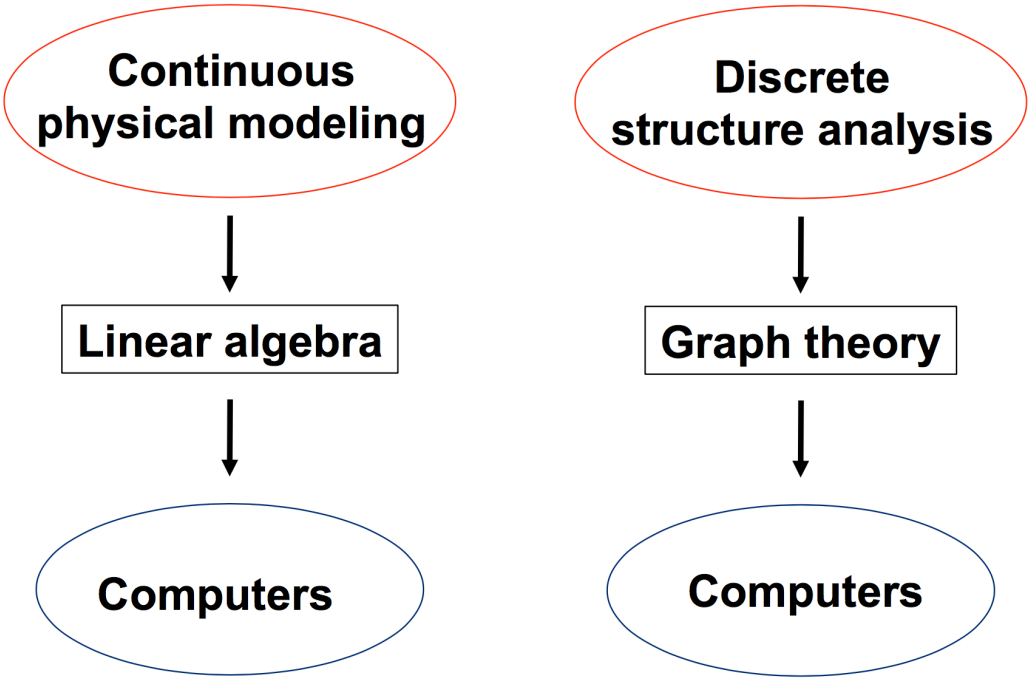
with or whitout a driving terms. This give you waves (radio, light and now even gravity!). Harder because it oscilalte like our pendulum problem! (By the way with an i front of the diffusion (or parabolic equation) it become the Schoedinger wave equation! The factors of i are all important as we say with the oscillator.

Another important theme is the relation between continuum (i.e calculus) and discrete (i.e. numerical) methods. They are of course intimately related.

Even at the **discrete** level the central role of Graphs is in fact identical to the linear algebra of Matrices. See for example the text see the book (it isn't easy stuff) on

Graph Algebra in the Language of Linear Algebra on github by Jeremy Kepner and John Gilbert. Also there are sldes John Gibert with pretty pictures to give you an overview.

The second important theme is that aboth PDEs and Linear Algebra come from minimization of quadratic (forms). Finally FEM says start with continuum, approxmate it with "flat" surfaces and voila you get the matrix equation. This is a fancy and powerful way of turning derivatives into differences that work in much more complicated geometries. VERY neat idea even if it is sort of obvious. (Most important ideas are obvious once someone hit on it!)



Here is a brief idea of the relation.

12.1 From Matrices to Graphs to Geometry

A Square matric is the connection/weight data of a general graph $G(A, N)$ The now square matrices

Coloring of a graph and paralellization.

12.2 Piecewise linear intropliationg and the demystification of FEM

See FEM page at (where else) Wikipedia:

https://en.wikipedia.org/wiki/Finite_element_method