

# EC526 Parallel Algorithm for High Performance Computing

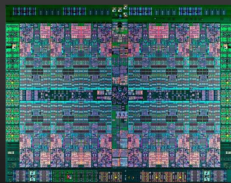
*Lecture Notes by Richard C. Brower et al*

Spring 2024: Tues, Thurs 130pm-3:15pm CDS 163

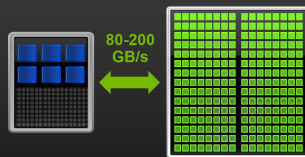
**Course Description:** The explosive advances in High Performance Computing, Big Data and Machine Learning require software design and parallel algorithms for distributed computing and data sets. Examples will be drawn from FFTs, Dense and Sparse Linear Algebra applied to simple physical systems using Multigrid Solvers, Monte Carlo Sampling and Finite Elements with a final team project 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 OpenACC threads on GPUs. Rapid prototypes and graphics methods will be provided with Mathematica. (Instructor: Prof. Richard Brower )



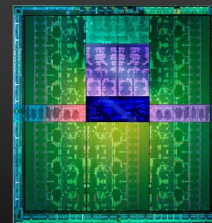
## Accelerated Computing 5x Higher Energy Efficiency



**IBM POWER CPU**  
Most Powerful Serial Processor



**NVIDIA NVLink**  
Fastest CPU-GPU Interconnect



**NVIDIA Volta GPU**  
Most Powerful Parallel Processor

**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”. Each student will pick one dwarf and work in a team to parallelize and optimization on a high performance computer. (Prerequisite: Some basic of C programming at the level of EC327 or EC602. All other software tools will be taught and example programs provided on github.)

# Course Syllabus

This is an over complete list. **Time constraints and interest of students** will select and deselect topics that are included in coding exercises.

## 1. First Part Weeks 1-4 : **Algebra**

- Intro to HPC and Engineering impact
- Language type: Symbolic, Interpreted and Compiled
- The ubiquitous Linear Algebra: HPC/ML/BigD/QC
- Numerical algebra for calculus and Floating Point Errors.
- Differentiation & Integration (Gauss and Monte Carlo)
- Newtons method for root finding & Non-linear optimization
- Data Analysis, Curve fitting & Error analysis
- FFT and related examples of recursion for divide and conquer
- Simple ODEs (oscillation vs relaxation)

## 2. Second Part: Weeks 5-9: **Parallelization Tools**

- PDE in Electrostatics and Image Processing,
- Select Projects & Parallelization Method
- OpenMP and OpenACC (for CUDA) threads.
- Message passing with MPI and Network Performance
- Scripts for submitting to HPC systems.
- Use of Numerical Libraries

## 3. Third Part Weeks 10-14 (5 weeks): **Special Topics/Projects**

- Dynamical PDEs for Heat Flow vs Wave propagation.
- Conjugate Gradient and iterative solvers
- Multi-grid Linear Solvers
- MD short range (neighborhood tables) vs long range Coulomb
- MonteCarlo for Magnetism: Cluster and Graphs
- Finite Element Method in 2D
- Digital Quantum Computing Programs

There is no Text – Unfortunately it hasn't been written in spite of the centrality of these fundamental algorithms and software tool in current trends in HPC, Big Data and Machine Learning. The course materials and exercises will be posted on [github](#). There are useful online documents that are referenced throughout the course and a few sections in books that are providing as a library in the Lab in PHO 207.

**Course Outcomes:** Students will be able to

- Develop efficient mathematical kernels, controlling round off and convergence.
- Develop data structures for basic linear operator on regular and semi-regular grids.
- Analyzed complexity and accuracy of linear solvers with FFT and Multi-scale methods.
- Implement parallel software with openMP or openACC (for CUDA) threads and MPI message protocols.
- Work in a team project on integrated software using Makefiles and github repositories.

# Course Organization and Grading

All students need to bring their laptops to class, which can be used for code development. Student will have accounts in **ec526** project on the [Shared Computer Cluster](#). This will give a uniform environment and more important access to modern High Performance hardware. Class participation and software development in class is a vital part of the course. The course will be run in the *open class room* style combining lectures and some time to work on software and and Q&A on the algorithmic methods.

Here is a basic break down of the grading.

**Software Exercises: 40 % of grade:** Every week to two weeks there will be a written Homework and a Laboratory software exercise. The grades will be based on the written answers and output of these programs in terms of performance (speed and accuracy) results presented in tables and graphs. Some exercises will include “pencil and paper” work on the algorithmic concepts. Code must be not be “copy and pasted” from on line examples and no sharing of code between students.

**Mid Term Exam: 20 % of grade:** There will be no final exam.

**Team Project: 30% of grade:** After the midterm term we will start the projects but hopefully teams and topics will be formed earlier. The presentation of the project at the end of the semester is in lieu of a final exam.

**In class Participation: 10% of grade:** The goal of in class work is to help with developing good coding methods in term of style organization, clarity and comments. Sharing ideas (not code) is encourage in class and on Slack. Readable code is useful especially in team software projects who of course do share code.

## 2022 Project Schedule (TO BE DETERMINED)

Schedule:

- March ? and Wednesday ? : Set up teams of 2-3 each.
- April ? : Present Team Project Plan: Who is responsible for what part. 4 slides.
- Wednesday April ? : Code and Written report: A separate one page on individual contribution and team assessment.
- May ? & ? : Final Project Presentation via Slides: 30 min maximum each tea

Project Requirements.

- Each team must write their own code and go in different directions. However sharing ideas during the Lectures is ok – in fact encouraged.
- Problem statement, tasks and goals with a 4 -5 slide proposal.
- Statement of the equations to solve and the basic algorithm.
- An assessment 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 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.** Each project should include some use of MPI and/or openMP to speed up the problem and a discussion of performance and accuracy. Part of the report can be library research on more advance methods that you might pursue if you were to take on this professionally!

# Contents

<b>1</b>	<b>Lecture Notes: Introduction</b>	<b>1</b>
1.1	What is High Performance Computing (HPC)? . . . . .	1
<b>2</b>	<b>Lecture Notes: Derivative to Finite Differences</b>	<b>3</b>
2.1	Higher order and Undetermined Coefficients . . . . .	4
2.2	Avoid Round off for $\Delta_h f(x)$ . . . . .	4
<b>3</b>	<b>Lecture Notes: Integrals to Sums</b>	<b>6</b>
3.1	Gaussian Integration . . . . .	7
<b>4</b>	<b>Lecture Notes: Parallelization Methods</b>	<b>11</b>
<b>5</b>	<b>Lecture Notes: Root Finding</b>	<b>12</b>
<b>6</b>	<b>Lecture Notes: Solving linear equation by Gaussian Elimination</b>	<b>14</b>
<b>7</b>	<b>Lecture Notes: Introduction to Matrices</b>	<b>17</b>
7.1	Matrices as Minimization of Quadratic Equations. . . . .	17
7.2	Solving the Matrix problem . . . . .	18
<b>8</b>	<b>Lecture Notes: Curve fitting &amp; Error analysis</b>	<b>26</b>
<b>9</b>	<b>Lecture Notes: FFT as recursion for divide and conquer</b>	<b>31</b>
9.1	Fast Fourier Transform . . . . .	31
9.2	Complex Numbers and Discrete Fourier Series . . . . .	32
9.2.1	FFT details . . . . .	33
9.3	Complex Fourier Test Code . . . . .	35
<b>10</b>	<b>Lecture Notes: Multigrid Recursion to Speed up Matrix Solvers</b>	<b>37</b>
10.1	1 D Laplacian Multigrid . . . . .	37
10.2	Fixed Boundary Conditions . . . . .	39
<b>11</b>	<b>Suggested Project Topic &amp; References</b>	<b>41</b>
11.1	Some ideas and References . . . . .	41
<b>12</b>	<b>Lecture Notes: Conclusions/Foward</b>	<b>44</b>
12.1	From Matrices to Graphs to Geometry . . . . .	45
12.2	Piecewise linear interpolation and the demystification of FEM . . . . .	45

# 1 Lecture Notes: Introduction

**WARNING - UNDER CONSTRUCTION** - These lecture notes are under continuous construction. The goal is to give the formal background to HPC software and algorithms. Also to provide links to the huge online documents Wikipedia resources. This is the real *text book* in a rapidly. The only way to have up to date resources in a rapidly developing field. The latest version will be maintained <https://github.com/brower/EC526.2022> with detailed software tools and coding instruction in the Exercises.

## 1.1 What is High Performance Computing (HPC)?

There is revolution underway with new “buzz words”: **Big Data** and **Machine Learning** that makes this question timely. There is a big debate going on to try to answer this! No matter the answer this is an explosively growing field and HPC expertise will be redefined and adapted to realize the promise.

Traditionally HPC was consider as a crucial, highly specialized but somewhat separate discipline. The singular goal was to harness the highest available hardware systems – always refereed as **Super Computers** – to explore physical science, predict new phenomena and guide engineering/industrial design. Conference on Super Computing:

(See last SC2019 <https://sc19.supercomputing.org> and next SC2020 <https://sc20.supercomputing.org> )

The list of the Top 500 (<https://www.top500.org> machines world wide tracked the exponential growth in hardware and algorithmic efficiency. The special characteristic were massively parallel algorithm, high speed networks and scaling to large data sets.

But the current revolution has made all these feature common to a larger and general field. As clock speeds hit a wall due to energy and cooling constraints **all** advanced hardware is parallel. Even you laptop has multi-core CPUs and even Graphics chips. The big data revolution requires large distributed memory system and machine learning clever adaptive algorithms. All these are feature are present in traditional HPC. Of course new application do require new methods but they are or should most often be built on HPC experience. By all means invent but “don’t reinvent the wheel”. These notes are designed to provide the wheels.

The US is one of several countries racing to get to Exascale computing ECP (<https://www.exascaleproject.org>).

$$\text{Exaflops} = 1,000,000,000,000,000 \text{ floating point operation per second} \quad (1)$$

Current pre-Exaflop Leader: Summit IBM/NVIDA machine at Oak Ridge: <https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/>

My Science: <https://www.olcf.ornl.gov/2019/07/31/alcc-program-awards-6-million-hours-on-olcf-resources/>

The next possible revolution: Quantum Computing <https://www.ibm.com/quantum-computing/network/overview/>

General Discussion and Introduction

- What is a computer?: Read about C template library and Ada Lovelace’: Augusta Ada King, Countess of Lovelace was a British mathematician and writer, chiefly known for her work on Charles Babbage’s



early mechanical general-purpose computer, the Analytical Engine.  
<https://www.sdsc.edu/ScienceWomen/lovelace.html>

- Who am I? Quantum Field Theory & The Exascale Revolution & Quantum Computing ?
- What are High Performance Algorithms:
- Applications and Motifs.
- From Continuum to Discrete and Back: Applicable Math/Software
- Basic Math Methods: **Guess and Verify!**

Some background reference and reading to explore in your spare time!

- William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, Numerical Recipes, 2nd ed., Cambridge University Press (1993). (free on line: <http://apps.nrbook.com/c/index.html>)
- Knuth: Concrete Mathematics ( [https://en.wikipedia.org/wiki/Concrete\\_Mathematics](https://en.wikipedia.org/wiki/Concrete_Mathematics) )
- On line C++: ( <http://www.cplusplus.com/doc/tutorial/> )
- On line (<http://www.gnuplot.info>) Piping from C: <http://stackoverflow.com/questions/29730732/piping-data-to-gnuplot-from-c>
- On line alpha Mathematica: (<http://www.wolframalpha.com>)
- CUDA demos  
<https://developer.nvidia.com/cuda-code-samples> (run at  
`/ad/eng/support/software/linux/all/x86_64/cuda/cuda_sdk-4.1/C/bin/linux/release`) Try em  
smokeParticles, particles, fluidsGL, nbody, ocieanFFT

Invitation to others to suggest interesting late night reading.

## 2 Lecture Notes: Derivative to Finite Differences

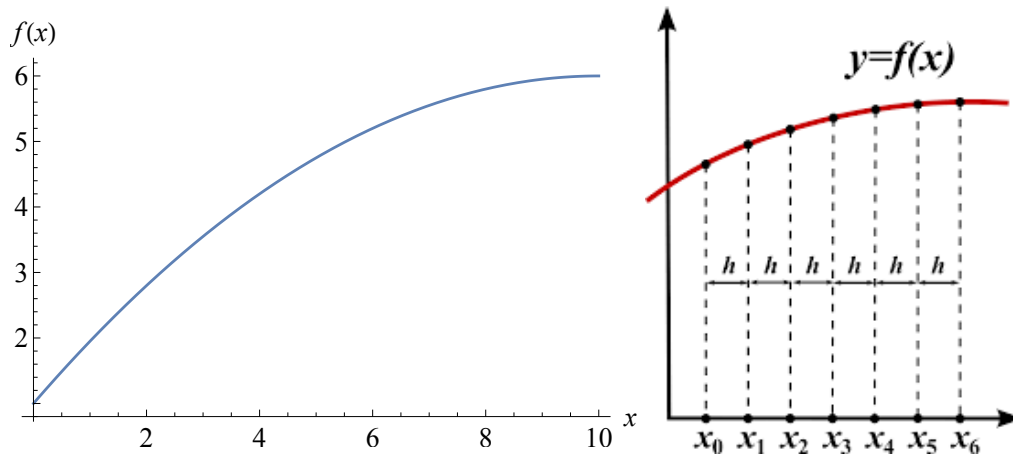


Figure 1: On left a function of  $x$ . On right the discrete points evaluated on a grid

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)

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots \quad (2)$$

What is  $a_0$ ? What is  $a_1$ ? Hint:

$$\frac{dx^n}{dx} = nx^{n-1} \quad \text{or map} \quad Dx^n \rightarrow nx^{n-1} \quad (3)$$

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  $f''(0) = 2a_2$  etc. Ok now we have “derived” the Taylor series!

$$f(x) = f(0) + xf'(0) + \frac{1}{2!}x^2f''(0) + \dots + \frac{1}{n!}x^nf^{(n)}(0) + \dots \quad (4)$$

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 )

$$\Delta_h f(x) = \frac{f(x+h) - f(x)}{h} \quad \text{and} \quad \tilde{\Delta}_h f(x) = \frac{f(x) - f(x-h)}{h} \equiv -\Delta_{-h} f(x) \quad (5)$$

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 would write,

$$\Delta f(x) = f(x+1) - f(x) \quad \text{and} \quad \tilde{\Delta} f(x) = f(x) - f(x-1) \quad (6)$$

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



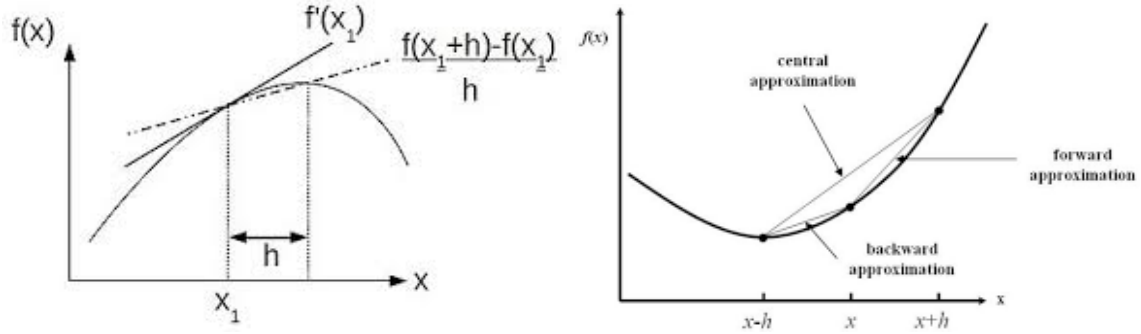


Figure 2: On left, finite difference. On right, central difference.

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

## 2.1 Higher order and Undetermined Coefficients

Higher order approximations add  $(\Delta_h + \tilde{\Delta}_h)/2$  and  $(\Delta_{2h} + \tilde{\Delta}_{2h})/2$  with the appropriate weights to cancel  $O(h^4)$  term.

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

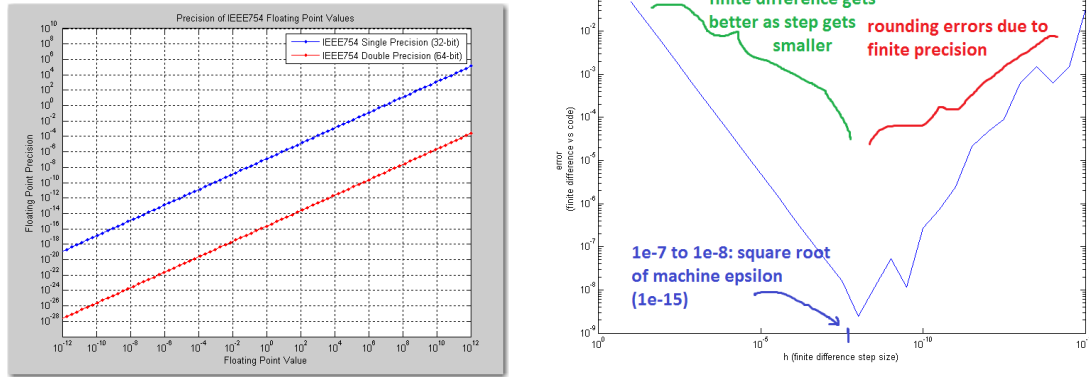


Figure 3: On the Right the error in the finite difference approximation to a derivative (y-axis) relative to size set size  $h$  (x-axis).

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

[https://en.wikipedia.org/wiki/Single-precision\\_floating-point\\_format](https://en.wikipedia.org/wiki/Single-precision_floating-point_format)

Consider the approximate derivative

$$df(x)/dx \simeq \Delta_h f(x) = \frac{f(x+h) - f(x)}{h} \quad (7)$$

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,

$$\begin{aligned}\Delta_h x^N &= [x^N + Nhx^{N-1} + \dots h^N - x^N]/h \\ &= Nx^{N-1} + \frac{N(N-1)}{2}hx^{N-1} + \dots + h^{N-1}.\end{aligned}\quad (8)$$

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

$$C[n][i] = \frac{n!}{(n-i)!i!} \quad \text{for } i = 0, 1, \dots, n \quad (9)$$

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:

$$C[n][i] = \begin{array}{c|cccccc} n \backslash i = & 0 & 1 & 2 & 3 & 4 & \dots \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & \dots \\ 1 & 1 & 1 & 0 & 0 & 0 & \dots \\ 2 & 1 & 2 & 1 & 0 & 0 & \dots \\ 3 & 1 & 3 & 3 & 1 & 0 & \dots \\ 4 & 1 & 4 & 6 & 4 & 1 & \dots \\ 5 & 1 & 5 & 10 & 10 & 5 & \dots \\ \dots & \dots & & & & & \end{array} \quad (10)$$

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

$$\Delta_h x^N = \sum_{i=1}^N C[N][i] x^{N-1} h^{i-1} = Nx^{N-1} + \frac{N(N-1)}{2!} x^{N-2} h + \dots + h^{N-1} \quad (11)$$

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.)

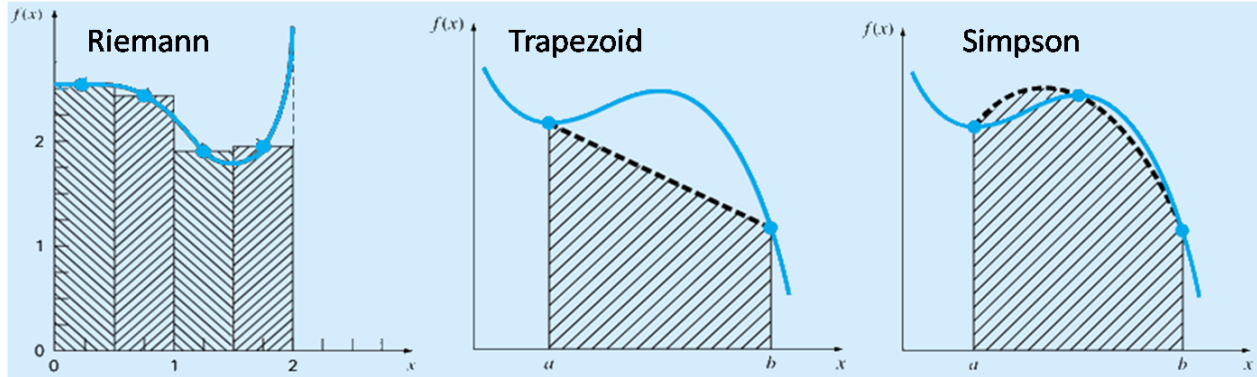
Similarly we can define  $\tilde{\Delta}_h$  as an explicit expansion of  $\tilde{\Delta}_h$ . What is? (Hint: Think what happen if  $h \rightarrow -h$ .) So the central difference cancels all term odd in  $h$ .

$$\frac{1}{2}(\Delta_h + \tilde{\Delta}_h)x^N \quad (12)$$

This is good for any finite difference  $\frac{1}{2}(\Delta_h + \tilde{\Delta}_h)f(x)$  of course so the central difference has its first error in  $O(h^2)$ . See notes and code at <https://github.com/brower/EC526>

### 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 trapezoidal 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

$$Dy(x) = \frac{dy}{dx} = f(x) \implies y(x) = \int_0^x f(x')dx' + c. \quad (13)$$

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

$$D^{-1}f(x) = g(x) = \int_0^x f(x')dx' + c \quad (14)$$

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 analogous expression:

$$\Delta_h y(x) = f(x) \implies y(Nh) = \sum_{i=0}^{N-1} f(ih)h + c; \quad (15)$$

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

$$\Delta^{-1}f(x) = f(x-1) + f(x-2) + \dots + f(0) + c \quad (16)$$

Is this right? Let's check it.

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

All terms cancel except the first, if we take  $f(-1) = 0$  as a convention and so we could add it to “definition” of  $\Delta^{-1}$ . 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 convenient to avoid cluttering 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:

$$\begin{aligned}
\Delta_h \Delta_h^{-1} f(x) &= \Delta[f(x-h) + f(x-2h) + \cdots + f(0) + f(-h) + c]h \\
&= (f(x) - f(x-h)) + (f(x-h) - f(x-2h)) + \cdots + f(0) - f(-h) \\
&= f(x)
\end{aligned} \tag{18}$$

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

$$I = \int_b^a f(x) dx \rightarrow I_N = \sum_{i=1}^N w_i f(x_i) \tag{19}$$

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  $(x_0, x_1, \dots, x_N)$

$$\begin{aligned}
y &= \sum_{i=0}^{N-1} h[f(a+ih) + f(a+(i+1)h)]/2 \\
&= h[\frac{1}{2}f(x_0) + f(x_1) + f(x_2) \cdots f(x_{N-1}) + \frac{1}{2}f(x_N)]
\end{aligned} \tag{20}$$

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

$$\int_{x_1}^{x_2} f(x) = h[f_1 + f_2]/2 + O(h^3 f'') \tag{21}$$

Exact for any  $f(x) = c_0 + c_1 x$  of course. With two free "weights," we can make any two terms exact:  $\int_{-h}^h f(x) = h[c_{-1}f(-h) + c_1f(h)]$  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 interval and coefficients symmetrically around  $x = 0$ . This implies

$$y_N = \sum_{i=0}^{N-1} (x_{i+1} - x_i) \frac{f(x_i) + f(x_{i+1})}{2} \tag{22}$$

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:

$$\int_{-h}^h f(x) dx = h \left[ \frac{1}{3}f(-h) + \frac{4}{3}f(0) + \frac{1}{3}f(h) \right] + O(h^5 f^{(4)}) \tag{23}$$

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

$$\begin{aligned}
2 &= c_{-1} + c_0 + c_1 \\
(2/3)h^3 &= h^3(c_{-1} + c_1)
\end{aligned} \tag{24}$$

### 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

$$\int_{-1}^1 f(x) dx \simeq \sum_{i=1}^N w_i f(x_i) \tag{25}$$

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  $2N$  knobs to give a perfect answer to all the powers:  $1, x, x^2, \dots, x^{2N-1}$  – 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](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. maximum power is  $x^N$ ), whose zeroes just happen 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<sup>1</sup>. Even the cubic is not easy (see [https://en.wikipedia.org/wiki/Cubic\\_function](https://en.wikipedia.org/wiki/Cubic_function)).

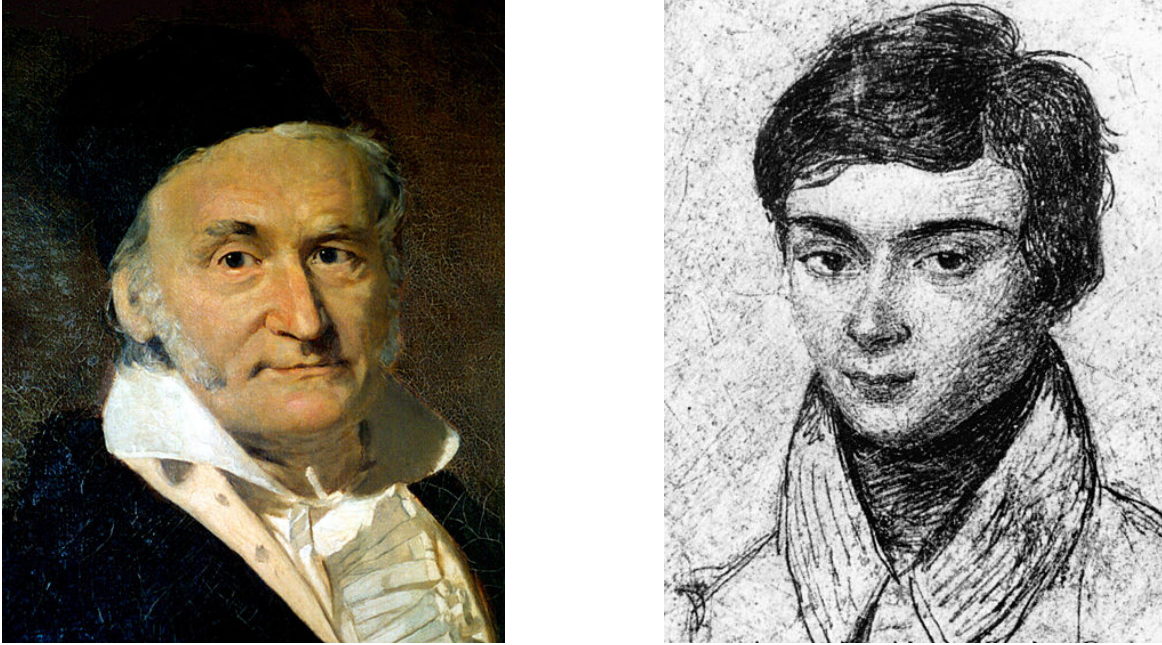


Figure 4: On left Gauss of course. On right Galois – whom lost a dual at age 22!

As a quick aside: thus far, we've only been concerning ourselves with intervals  $-h < x < h$ , or  $-1 < x < 1$ . In an integral you always change variable to make it easier. Here with the linear change of the variable,  $x' = (b-a)x/2 + (b+a)/2$ , we can transform any arbitrary interval  $a < x < b$  into  $-1 < x < 1$  by shifting and the rescaling:

$$\int_a^b f(x') dx' = \frac{(b-a)}{2} \int_{-1}^1 f\left(\frac{b-a}{2}x + \frac{b+a}{2}\right) dx \quad (26)$$

Now  $x = 0$  corresponds to the mid point  $x' = (a+b)/2$  and with scaling the end points:  $x = -1 \leftrightarrow x' = a$  and  $x = 1 \leftrightarrow x' = b$ .

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  $x$ )

$$I_1 = w_0 f(0) \quad (27)$$

so

$$2 = w_0 \quad \text{for } 1 \quad (28)$$

This is also known as midpoint integration:

$$I_1 = 2f(0), \quad (29)$$

---

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

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

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

$$I_2 = w_1 f(-\delta) + w_1 f(\delta) \quad (30)$$

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:

$$\begin{aligned} 2 &= w_1 + w_1 & \text{for } 1 \\ 2/3 &= w_1 \delta^2 + w_1 \delta^2 & \text{for } x^2 \end{aligned} \quad (31)$$

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  $x^5$ )

$$I_3 = w_1 f(-\delta) + w_0 f(0) + w_1 f(\delta) \quad (32)$$

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

$$\begin{aligned} 2 &= w_1 + w_0 + w_1 & \text{for } 1 \\ 2/3 &= w_1 \delta^2 + w_1 \delta^2 & \text{for } x^2 \\ 2/5 &= w_1 \delta^4 + w_1 \delta^4 & \text{for } x^4 \end{aligned} \quad (33)$$

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

$$\int_{-1}^1 f(x) dx \simeq \frac{1}{9} [5f(-\sqrt{3/5}) + 8f(0) + 5f(\sqrt{3/5})] \quad (34)$$

or

$$\begin{aligned} \int_{-a}^b f(x') dx' &= \frac{(b-a)}{18} \left[ 5f \left( -\frac{(b-a)}{2} \sqrt{3/5} + \frac{(b+a)}{2} \right) \right. \\ &\quad \left. + 8f \left( \frac{(b+a)}{2} \right) + 5f \left( \frac{(b-a)}{2} \sqrt{3/5} + \frac{(b+a)}{2} \right) \right] \end{aligned} \quad (35)$$

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:

$$w_i = \frac{2(1 - x_i^2)}{[(n+1)P_{n+1}(x_i)]^2} \quad (36)$$

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

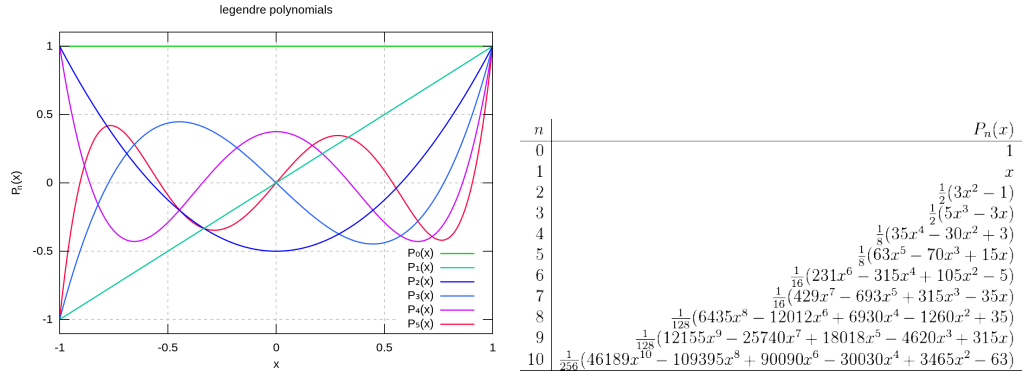


Figure 5: On left, the plots of the Legendre Polynomial listed on the right.



## 4 Lecture Notes: Parallelization Methods

**LARGE Under construction**

Discussion of parallelizable and non-parallelizable code structure.

Basic role of Threads, Open MP, on GPU with OpenACC vs Message Passing in MPI. Hybrid Models with mix and match.

## 5 Lecture Notes: Root Finding

In **Numerical Recipes in C** at <http://apps.nrbook.com/c/index.html> in Chapter 9 there are nice discussions 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.

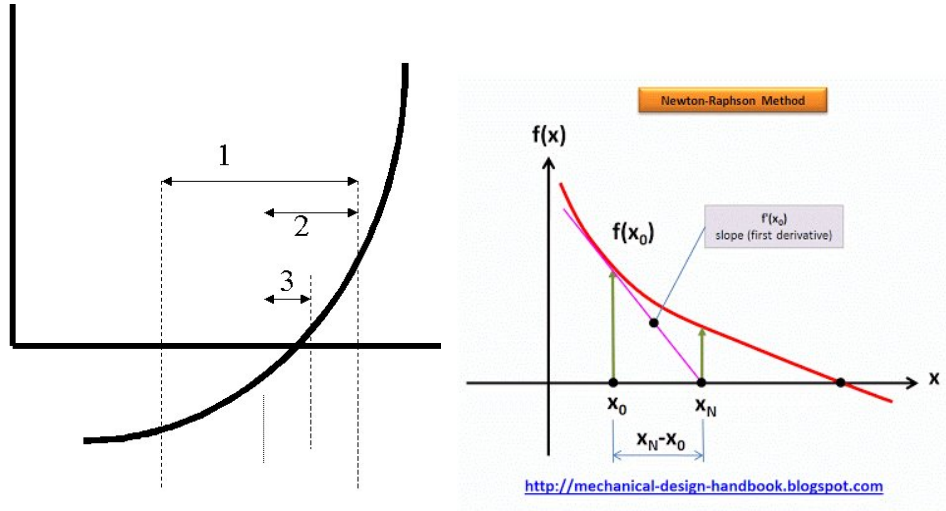


Figure 6: On left root finding by bisection. On the right root finding by Netwon-Raphson.

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  $f(x_c \equiv \frac{x_{min} + x_{max}}{2})$ . 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  $f(x)$ ,

$$f(x) \simeq f(x_0) + (x - x_0)f'(x_0), \quad (37)$$

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

$$x \leftarrow x - f(x)/f'(x). \quad (38)$$

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:

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

so the computer does the iteration,

$$x = \frac{1}{2}\left(x + \frac{A}{x}\right). \quad (39)$$

The fixed point is  $x = A^{1/2}$ , because  $A^{1/2} = \frac{1}{2}(A^{1/2} + A/A^{1/2})/2 = A^{1/2}$ . While bisection is  $O(\log_2(N))$  to get N-bit accuracy Newton's method is  $O((\log_2(\log_2(N)))$  – like the comparison for bisection vs dictionary method for find algorithm in a sorted list of N item.

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

## 6 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](https://en.wikipedia.org/wiki/Gaussian_elimination)

As an illustration let's consider a system of equations mercilessly stolen from the internet<sup>2</sup>:

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 2x + y + 3z &= 0 \\ 4x - 3y + z &= -10 \end{aligned} \tag{40}$$

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 explicit 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

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 5y - 7z &= 34 \\ 5y - 19z &= 58 \end{aligned} \tag{41}$$

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:

$$\begin{aligned} -x + 2y - 5z &= 17 \\ 5y - 7z &= 34 \\ -12z &= 24 \end{aligned} \tag{42}$$

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:

$$\begin{array}{lll} -x + 2y - 5z = 17 & -x + 2y + 10 = 17 & -x + 2y = 7 \\ 5y - 7z = 34 & \implies 5y + 14 = 34 & \implies 5y = 20 \\ z = -2 & z = -2 & z = -2 \end{array} \tag{43}$$

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:

$$\begin{array}{lll} -x + 2y = 7 & -x + 8 = 7 & -x = -1 \\ y = 4 & \implies y = 4 & \implies y = 4 \\ z = -2 & z = -2 & z = -2 \end{array} \tag{44}$$

---

<sup>2</sup><http://www.sparknotes.com/math/algebra2/systemsofthreeequations/section1.rhtml>

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

$$\begin{aligned}x &= 1 \\y &= 4 \\z &= -2\end{aligned}\tag{45}$$

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

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

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:

$$\begin{aligned}4x - 3y + z &= -10 \\-x + 2y - 5z &= 17 \\5y - 7z &= 34\end{aligned}\tag{47}$$

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:

$$\begin{aligned}-x + 2y - 5z &= 17 \\2x + y + 3z &= 0 \\4x - 3y + z &= -10\end{aligned}\tag{48}$$

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:

$$\begin{bmatrix} -1 & 2 & -5 & 17 \\ 2 & 1 & 3 & 0 \\ 4 & -3 & 1 & -10 \end{bmatrix}\tag{49}$$

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:

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

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.

## 7 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 Vectors/Matrices in numerical linear algebra.

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

Most algorithms clear correspondence as well.

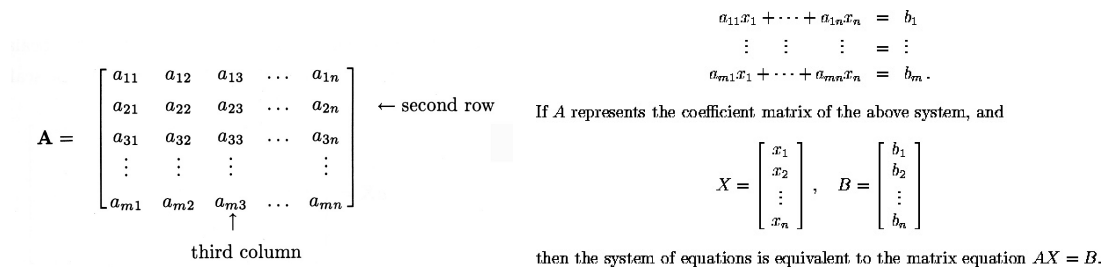


Figure 7: On left .....On the right ....

The  $\mathbf{Ax} = \mathbf{b}$  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

$$\sum_j a_{ij} x_j = b_i \quad \Longleftrightarrow \quad AX = B \quad (50)$$

## 7.1 Matrices as Minimization of Quadratic 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

$$f(x) = \frac{1}{2}[(ax - b)^2 + c] = \frac{1}{2}[a^2x^2 - 2abx + b^2 + c] \quad (51)$$

Then minimum has zero derivative,

$$\frac{df(x)}{dx} = a(ax - b) = 0 \quad \text{or} \quad x = b/a \quad (52)$$

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

$$S[x_1, \dots, x_N] = \frac{1}{2} \sum_i (\sum_j a_{ij} x_j - b_i)^2 \quad (53)$$



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

$$\frac{\partial S}{\partial x_k} = \sum_i a_{ik} \left( \sum_j a_{ij} x_j - b_i \right) = 0 \quad (54)$$

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

$$a_{ik}(a_{ij}x_j - b_i) = 0 \iff A^T(Ax - b) = 0 \quad (55)$$

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

$$Ax = b \quad (56)$$

if we have too few variable we need to solve,

$$A^T Ax = A^T b \quad (57)$$

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

$$\chi^2 = \sum_{i=0}^{N_0} \frac{(y_i - f(x_i))^2}{\sigma_i^2} = (y_i/\sigma_i - (x_i^n/\sigma_i)c_n) \quad (58)$$

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

$$A^T A c = A^T b \iff \sum_i (x_i^{m+n}/\sigma_i^2) c_n = \sum_i x_i^m y_i / \sigma_i^2 \quad (59)$$

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 minimization when you don't have enough variables!

## 7.2 Solving the Matrix problem

(Google Laplace Smoother and Graph Laplacian et al)

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  $x, y \in \{0, 1, \dots, L-1\}$ , 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:

$$\text{px}[x][y] = 0.25 * (\text{px}[(x+1)\%L][y] + \text{px}[(x-1+L)\%L][y] + \text{px}[x][(y+1)\%L] + \text{px}[x][(y-1+L)\%L]);$$

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:

$$\begin{aligned} \text{px}[x][y] = (1-r) * \text{px}[x][y] + r/4 * (\text{px}[(x+1)\%L][y] + \text{px}[(x-1+L)\%L][y] \\ + \text{px}[x][(y+1)\%L] + \text{px}[x][(y-1+L)\%L]); \end{aligned}$$

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:

$$\begin{pmatrix} 0 & r/4 & 0 \\ r/4 & 1-r & r/4 \\ 0 & r/4 & 0 \end{pmatrix} \quad (60)$$

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  $(1-r) * \text{px}[x][y]$ . The rightmost  $r/4$  corresponds to the location  $(1,0)$ , which can be thought of as the contribution of the pixel to the right:  $r/4 * (\text{px}[(x+1)\%L][y])$ . The rest of the elements contribute analogously—the weighted average above can be completely encoded by this stencil (up to the assumption of periodic boundary conditions).

This is special case of a connection between **smoothing** with a discrete Laplacian and a general concept in graph theory of the Graph Laplacian: [https://en.wikipedia.org/wiki/Laplacian\\_matrix](https://en.wikipedia.org/wiki/Laplacian_matrix) For fun see

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. <sup>3</sup>

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  $\text{px}$  as a function  $f$ , which we'll give some possible physical interpretations in a bit. The operation becomes:

$$f(x, y, t+1) = (1-r)f(x, y, t) + \frac{r}{4} [f(x+1, y, t) + f(x-1, y, t) + f(x, y+1, t) + f(x, y-1, t)] \quad (61)$$

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  $\pm 1$  because we were thinking of sites on a grid; let's give it some

<sup>3</sup>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.

physical distance instead, which we'll call  $a$ . Similarly, we'll replace the  $+1$  in time with  $+\delta t$ , for a little time 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:

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

where on the last line I implicitly defined

$$\nabla^2 = \frac{d^2}{dx^2} + \frac{d^2}{dy^2} \quad (62)$$

as the *Laplacian*. The Laplacian is denoted in some other texts as  $\Delta$ . 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  $\delta t$ . 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 diffusion 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]);
        }
    }
    // Copy Tnew to T
    done = 1;
}

```

```

    }
  }
  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.

## 8 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:

$$y = mx + b \tag{63}$$

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:

$$L_1(x) \propto (x - 3) \tag{64}$$

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  $x = 1$ .

$$L_1(x) \propto \frac{(x - 3)}{(1 - 3)} \tag{65}$$

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:

$$L_1(x) = 2 \frac{(x-3)}{(1-3)} \quad (66)$$

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:

$$\begin{aligned} L_2(x) &\propto (x-1) \\ &\propto \frac{(x-1)}{(3-1)} \\ &= 4 \frac{(x-1)}{(3-1)} \end{aligned} \quad (67)$$

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

$$\begin{aligned} y &= L_1(x) + L_2(x) \\ &= 2 \frac{(x-3)}{(1-3)} + 4 \frac{(x-1)}{(3-1)} \end{aligned} \quad (68)$$

One can quickly verify that the line is 2 at  $x = 1$ , 4 at  $x = 3$ . We can also simplify it to 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  $(x_i, y_i)$ ,  $i = 1, \dots, n + 1$  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:

$$L_1(x) \propto (x-9)(x-4) \quad (69)$$

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  $x = 1$ .

$$L_1(x) \propto \frac{(x-9)(x-4)}{(1-9)(1-4)} \quad (70)$$

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:

$$L_1(x) = 5 \frac{(x-9)(x-4)}{(1-9)(1-4)} \quad (71)$$

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

$$y = L_1(x) + L_2(x) + L_3(x) \quad (72)$$

$$= 5 \frac{(x-9)(x-4)}{(1-9)(1-4)} + 2 \frac{(x-1)(x-4)}{(9-1)(9-4)} + 1 \frac{(x-1)(x-9)}{(4-1)(4-9)} \quad (73)$$

$$= \frac{23}{120}x^2 - \frac{55}{24}x + \frac{71}{10} \quad (74)$$

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 ≠ i do
        | g(x)* = (x - xj)/(xi - xj)
    end
    f(x)+ = yi 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  $\chi^2$  fitting: derive average.
- Segway for mean, error, etc.
- General  $\chi^2$ .

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  $y_i$ ,  $i = 1, 2, \dots, N$ , optionally with standard errors  $\sigma_i$  (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:

$$f(x; a_0, \dots, a_m) = a_0 + \sum_{i=1}^m a_i x^i, \quad (75)$$

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:

$$\left( \begin{array}{ccccc|c} \sum_{i=1}^N \frac{1}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^2}{\sigma_i^2} & \cdots & \sum_{i=1}^N \frac{x_i^m}{\sigma_i^2} & \sum_{i=1}^N \frac{y_i}{\sigma_i^2} \\ \sum_{i=1}^N \frac{x_i}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^2}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^3}{\sigma_i^2} & \cdots & \sum_{i=1}^N \frac{x_i^{m+1}}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i y_i}{\sigma_i^2} \\ \sum_{i=1}^N \frac{x_i^2}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^3}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^4}{\sigma_i^2} & \cdots & \sum_{i=1}^N \frac{x_i^{m+2}}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^2 y_i}{\sigma_i^2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sum_{i=1}^N \frac{x_i^m}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^{m+1}}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^{m+2}}{\sigma_i^2} & \cdots & \sum_{i=1}^N \frac{x_i^{2m}}{\sigma_i^2} & \sum_{i=1}^N \frac{x_i^m y_i}{\sigma_i^2} \end{array} \right) \quad (76)$$

Ok how do you estimate the value of sigma. Suppose you have measure set of height ( $h_1, h_2, \dots, h_N$ ) for a give size (with in our case different permutations) you can not only estimate average (or mean *overline{h}*!) but the error (or standard deviation  $\sigma$ ) by the well know formulae:

$$\bar{h} = \frac{1}{N} \sum_{i=1}^N h_i \quad , \quad \sigma^2 = \frac{1}{N} \sum_{i=1}^N (h_i - \bar{h})^2 \quad (77)$$

The mean and the width are the first and second moment that fix the form of a Gaussian (or Normal) distribution. (Gnuplot lets use  $\sigma$  in your plots and fits.) Of course as we discussed in class the distribution

may not be – in fact almost never is – a perfect Gaussian. Higher moments called **Skewness** and **Kurtosis** look for non-Gaussian features. For fun see:

<https://www.itl.nist.gov/div898/handbook/eda/section3/eda35b.htm>

## 9 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 example is the simple pendulum. In small oscillation it has single harmonic motion used in grandfather clocks to keep time. There is a cute Mathematical Notebook on github to understand the single pendulum 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 oscillators. 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\\_\(book\)](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 <sup>4</sup> Almost anything will oscillate around a stable point with **exactly** the same equation. Realistically need to add some damping or viscous drag to “ $F = ma$ ”:

$$m \frac{d^2 x}{dt^2} = F_{spring} = -kx \quad \text{and damping} \quad \frac{d^2 x}{dt^2} = F_{drag} = -\Gamma \frac{dx}{dt} \quad (78)$$

Recall the second equation if exponential  $x = x_0 e^{-\Gamma t}$  if try  $x = x_0 \exp[\lambda t]$  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:

$$x = x_0 e^{-\gamma t} e^{\pm i \sqrt{\omega_0^2 - \gamma^2} t} \quad (79)$$

For an LRC circuit  $x \rightarrow Q, m \rightarrow L, k \rightarrow 1/C, \Gamma \rightarrow R, F \rightarrow V$ . Non-linearities come in all sorts of forms! Pendulum  $x \rightarrow \theta, m \rightarrow I, F \rightarrow gl \sin(\theta)$ . 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 oscillator at [https://en.wikipedia.org/wiki/Harmonic\\_oscillator](https://en.wikipedia.org/wiki/Harmonic_oscillator) Circuits simulator (analogue computers) and digital computer are essential!

### 9.1 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 Gauss' earlier application: [https://en.wikipedia.org/wiki/Fast\\_Fourier\\_transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform)

Gauss, Carl Friedrich, "Theoria interpolationis methodo nova tractata",  
Werke, Band 3, 265327 (Königliche Gesellschaft der Wissenschaften, Göttingen, 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”  $y_n = f(t_k n)$  for  $n = 0, 1, 2, \dots, N-1$  for times at equal intervals  $t_n = n\Delta t$ . The Fourier series for a discrete sample of a “time” sequence with  $N$  terms  $n = 0, 1, \dots, N-1$  can be expanded in a series of sines and cosines:

$$y_n = a_0 + \sum_{k=1}^{N_0} [a_k \cos(2\pi kn/N) + b_k \sin(2\pi kn/N)] \quad (80)$$

<sup>4</sup>See LRC circuits: [https://en.wikipedia.org/wiki/RLC\\_circuit](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 that this fit in fact assume it is good to approximate the signal can be assumed to be (approximately) periodic for  $t \rightarrow t + N$  with frequencies:  $\omega_k = 2\pi k/N$ . (Warning: In our FT formalism we will use “time” to “frequency” with  $t_n \rightarrow \omega_k$  interchangeably with “space” to “wave number” with  $x \rightarrow k$ . Both of course use the same formulae.)

## 9.2 Complex Numbers and Discrete Fourier Series

New complex arithmetic. Special case

$$\cos(\theta) + i \sin(\theta) \equiv e^{i\theta} \quad (81)$$

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:

$$\cos(\alpha + \beta) + i \sin(\alpha + \beta) = (\cos \alpha \cos \beta - \sin \alpha \sin \beta) + i(\cos \alpha \sin \beta + \sin \alpha \cos \beta) \quad (82)$$

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

$$\cos(\theta) = \frac{e^{i\theta} + e^{-i\theta}}{2}, \quad \sin(\theta) = \frac{e^{i\theta} - e^{-i\theta}}{2i} \quad (83)$$

The expansion real Fourier expansion in Eq.80 exactly equivalent (and more more more conveniently written) as

$$y_n = \sum_{k=0}^{N-1} c_k e^{2\pi i k n / N} \quad (84)$$

This transforms  $N$  coefficients  $c_k$  into  $N$  values  $y_n = f(t_n)$  so is a linear equation ( $y_n = A_{nk} c_k \rightarrow c_k = A_{kn}^{-1} y_n$ ) that can in principle be solved. This is a set of linear equation in the coefficients. We could solve 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

$$c_k = \frac{1}{N} \sum_{n=0}^{N-1} y_n e^{-2\pi i k n / N} \quad (85)$$

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

$$\sum_n e^{2\pi i k n / N} e^{-2\pi i k' n / N} = \delta_{kk'} \quad (86)$$

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

We have seen one case of **divide and conquer** in searching a list using bisection. We start with an ordered 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 divide and conquer by bisection. The replace the number of step  $T(N) = \text{const}N$  to a recursive count that replaces  $T(N)$  by a half size problem with  $T(N/2)$  set. We can write an equation for this

$$T(N) = T(N/2) + c_0 \quad (87)$$

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  $N \rightarrow N/2 \rightarrow N/4 \dots \rightarrow 1$ . We can show that this works in this equation:

$$c \log(N) = c \log(N/2) + c_0 \quad (88)$$

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

### 9.2.1 FFT details

Let's look at this carefully. The Fourier transform <sup>5</sup> (expansion) is

$$y_k \equiv \mathcal{FT}_N[c_n] = \sum_{n=0}^{N-1} (\omega_N^n)^k a_n = \sum_{n=0}^{N-1} e^{i2\pi nk/N} a_n \quad (89)$$

The trick is use a binary representation of :  $n = n_0 + 2n_1 + 2^2n_2 + \dots + 2^pn_p$

$$\omega_N^n = \omega_N^{n_0} \omega_{N/2}^{n_1} \omega_{N/4}^{n_2} \dots \omega_2^{n_p} \quad (90)$$

$$\sum_n \omega_N^{nk} = \sum_{n_0=0,1} \omega_{N/2}^{n_1 k} \sum_{n_1=0,1} \omega_N^{n_0 k} \dots \sum_{n_p=0,1} \omega_2^{n_p k} \quad (91)$$

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

$$y_k = \sum_{n=0}^{N/2-1} e^{i2\pi nk/(N/2)} a_{2n} + \omega_N^k \sum_{n=0}^{N/2-1} e^{i2\pi nk/(N/2)} a_{2n+1} \quad (92)$$

Therefore, one  $N$  = two  $N/2$  Fourier transforms.

low k:

$$y_k = \sum_{n=0}^{N/2-1} e^{i2\pi nk/(N/2)} [a_{2n} + \omega_N^k a_{2n+1}] \quad (93)$$

high k:

$$y_{k+N/2} = \sum_{n=0}^{N/2-1} e^{i2\pi nk/(N/2)} [a_{2n} - \omega_N^k a_{2n+1}] \quad (94)$$

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

$$y_k = \sum_{n=0}^{N/2-1} e^{i2\pi nk/N} a_n + e^{i\pi k} \sum_{n=0}^{N/2-1} e^{i2\pi nk/N} a_{n+N/2} \quad (95)$$

Therefore, one  $N$  = two  $N/2$  Fourier transforms.

even k:

$$y_{2\tilde{k}} = \sum_{n=0}^{N/2-1} e^{i2\pi n\tilde{k}/(N/2)} [a_n + a_{n+N/2}] \quad (96)$$

---

<sup>5</sup>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  $1/\sqrt{N}$  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:

$$y_{2\tilde{k}+1} = \sum_{n=0}^{N/2-1} e^{i2\pi n\tilde{k}/(N/2)} \omega_N^n [a_n - a_{n+N/2}] \quad (97)$$

This pattern as a data flow is called the butterfly network:

It is a recursive pattern high vs low bit order is going right to left or left to right! Thus,

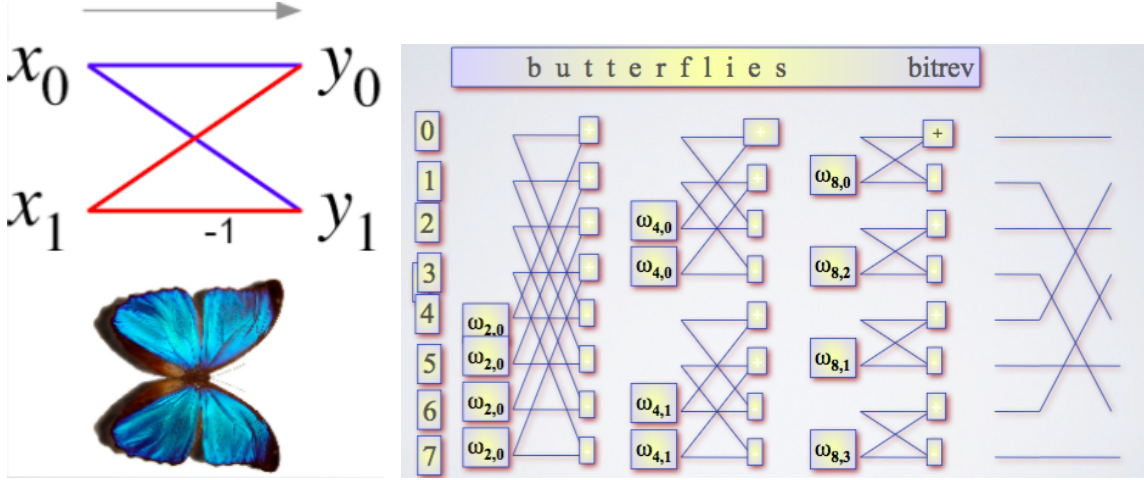


Figure 8: The is the so called butterfly network for the FFT in bit reversed form. The last step is to the right.  $\omega_{N,k} = \omega_N^n = (\omega_N)^n = e^{2\pi i k/N}$  above

$$T(N) = 2T(N/2) + c_0N \implies T(N) \sim N \log(N) \quad (98)$$

And in summary,

$$y_k = \mathcal{FT}_{N/2}[a_{2n} + \omega_N^k a_{2n+1}] \quad (99)$$

$$y_{k+N/2} = \mathcal{FT}_{N/2}[a_{2n} - \omega_N^k a_{2n+1}] \quad (100)$$

$$y_{2k} = \mathcal{FT}_{N/2}[a_n + a_{n+N/2}] \quad (101)$$

$$y_{2k+1} = \mathcal{FT}_{N/2} \omega_N^n [a_n - a_{n+N/2}] \quad (102)$$

Note however that the “bits are reversed” in the network flow.

### 9.3 Complex Fourier Test Code

Using complex variable 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(*Ftilde, 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
```

## 10 Lecture Notes: Multigrid Recursion to Speed up Matrix Solvers

Multigrid is a recursive approach to speeding up the  $Ax = b$  matrix solver. It shares ideas of “divide and conquer” like bisection searching or merge sorting AND ideas for Fourier 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 <sup>6</sup>. For the multigrid formalism it is crucial 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,

$$\phi_{out}[x] = A_h \phi[x] = \frac{2\phi[x] - \phi[x+h] - \phi[x-h]}{h^2} + m^2 \phi[x] \quad (104)$$

is equivalent to

$$A_h \phi = \frac{1}{h^2} \begin{bmatrix} 2 + h^2 m^2 & -1 & 0 & 0 & \cdots & 0 & -1 \\ -1 & 2 + h^2 m^2 & -1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 2 + h^2 m^2 & -1 & \cdots & 0 & 0 \\ 0 & 0 & -1 & 2 + h^2 m^2 & \cdots & 0 & 0 \\ \cdots & & & & & & \\ 0 & 0 & 0 & 0 & \cdots & 2 + h^2 m^2 & -1 \\ -1 & 0 & 0 & 0 & \cdots & -1 & 2 + h^2 m^2 \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \cdots \\ \phi_{L-2} \\ \phi_{L-1} \end{bmatrix} \quad (105)$$

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 awkward 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  $h \rightarrow 2h \rightarrow 2^2 h \cdots$  and lattice points  $L \rightarrow L/2 \rightarrow L/2^2 \cdots$ . Suppose after iterating a little you have an approximation  $\phi[x]$  to the exact solution  $\phi_{exact}[x]$

$$\phi_{exact}[x] = \phi[x] + e[x] \simeq \phi[x] . \quad (106)$$

There is still a small “residual” mismatch in our equation above

$$r[x] = b - A_{xy} \phi[y] \neq 0 \quad (107)$$

The error obeys the same matrix equation

$$A_h e = r \quad (108)$$

---

<sup>6</sup>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 solution so is  $\phi[x] + \text{const}$

$$-\frac{d^2 \phi(x)}{dx^2} \simeq \frac{2\phi[x] - \phi[x+h] - \phi[x-h]}{h^2} + m^2 \phi[x] = b[x] \quad (103)$$

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

1. The problem, approximated on fine lattice with lattice spacing  $h$  and  $x = 0, 1, 2, \dots, L-1$  sites, has error satisfying:

$$A_h e = r \quad (109)$$

2. We average (or project) residue on each block (even and odd pairs in 1D) to coarse sites  $\hat{x} = 0, 1, 2, \dots, L/2-1$ .

$$\hat{r}[\hat{x}] = \frac{r[2\hat{x}] + r[2\hat{x}+1]}{2} \quad (110)$$

3. Find the error on the  $2h$  grid using the coarse approximation,

$$\hat{A}\hat{e}[\hat{x}] = \hat{r}[\hat{x}] \quad (111)$$

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.

$$e[2\hat{x}] = \hat{e}[\hat{x}] \quad , \quad e[2\hat{x}+1] = \hat{e}[\hat{x}] \quad (112)$$

Now we have “guess” to the coarse operator,  $\hat{A}$ ? The obvious choice is to just look at Eq. , replace replacement:  $h \rightarrow 2$

$$\hat{A}\phi = A_{2h}\phi = \frac{2\phi[x] - \phi[x+2h] - \phi[x-2h]}{4h^2} + m^2\phi[x] \quad (113)$$

There is another popular form called the **Gelerken Prescription**: Instead we use the non-square matrices for projection  $P$  and interpolations  $Q$  defined by there action above as

$$\hat{r}[\hat{x}] = \sum_x P_{\hat{x},x} r[x] = \frac{r[2\hat{x}] + r[\hat{x}+1]}{2} \quad (114)$$

and

$$e[x] = \sum_{\hat{x}} Q_{x,\hat{x}} \hat{e}[\hat{x}] = \hat{e}[x/2] \quad (115)$$

respectively. The idea is to approximate the fine operator equation by

$$Ae = r \quad \rightarrow \quad (PAQ)Pe = Pr \quad (116)$$

as we take  $h \rightarrow 2h$ . The resulting Gerlerkin form defines  $\hat{A} = PAQ$  (or  $\hat{A}\hat{e} = \hat{r}$ ). A little tedious algebra gives,

$$\hat{A}\hat{e}(\hat{x}) = \frac{2\phi[x] - \phi[x+2h] - \phi[x-2h]}{2h} + m^2\phi[x] = b[x] \quad (117)$$

Strange! Only one factor of  $h$ ! in the first term! Not the naive scaling you might expect with a factor of  $(2h)^2$ . Both will work however.

We note in our code `mg.c` manipulated the quation into standard form

$$\hat{e}[\hat{x}] = \alpha(lev)(\hat{e}[\hat{x}+1] + \hat{e}[\hat{x}-1]) + \hat{r}[\hat{x}] \quad (118)$$

on the coarse lattices at levele level  $lev = 1, 2, \dots$  with spacing  $h \rightarrow 2^{lev}$ . 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

A nice illustration of this Multgrid cycle Fig.9 for a 2D lattice is taken form Wikipedia of course: [https://en.wikipedia.org/wiki/Multigrid\\_method](https://en.wikipedia.org/wiki/Multigrid_method).

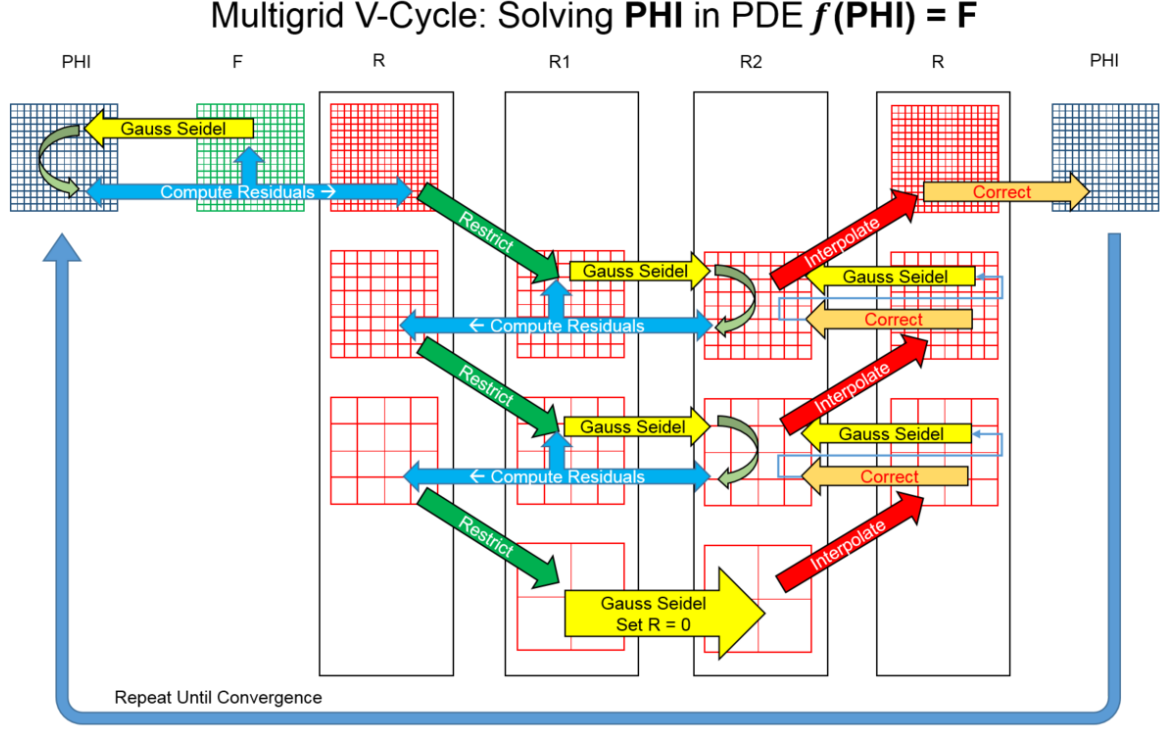


Figure 9: Visualization of iterative Multigrid algorithm for fast  $O(N)$  convergence.

## 10.2 Fixed Boundary Conditions

In some ways this is easier. We can drop the  $m^2$  term and consider an expanded interval  $x = -1, 0, 1, \dots, L - 1, L$  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  $\phi[x] \rightarrow T[x]$  for  $x = 0, 1, 2, \dots, L - 1$ . The matrix is  $L \times L$  acting on these points

$$AT[x] = \frac{T[x+1] + T[x-1] - 2T[x]}{h^2} + b_0[x] \quad \text{for } x = 0, 1, 2, \dots, L - 1 \quad (119)$$

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:

$$AT = \frac{1}{h^2} \begin{bmatrix} 2 & -1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 & 0 \\ 0 & 0 & -1 & 2 & \cdots & 0 & 0 \\ \cdots & & & & & & \\ 0 & 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & 0 & \cdots & -1 & 2 \end{bmatrix} \begin{bmatrix} T[0] \\ T[1] \\ T[2] \\ T[3] \\ \cdots \\ T[L-2] \\ T[L-1] \end{bmatrix} \quad (120)$$

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

$$b[x] = b_0[x] + \delta_{x,0}T_a/h^2 + \delta_{x,L-1}T_b/h^2 \quad (121)$$

include two contributions form the boundaries.



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:

$$r[x] = b[x] - AT[x] \quad \text{for } x = 0, 1, 2, \dots, L-1 \quad (122)$$

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  $x, y = 0, 1, 2, \dots, L-1$  in the interior and lines on the sides of the square: boudaries are lines

$$h^2 b[x][y] = h^2 b_0[x][y] + \delta_{x,0} T[-1][y] + \delta_{x,L-1} T[L][y] + \delta_{y,0} T[x][-1] + \delta_{y,L-1} T[x][L] \quad (123)$$

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

$$\hat{r}[\hat{x}][\hat{y}] = Pr[\hat{x}][\hat{y}] \frac{1}{4} (r[2\hat{x}][2\hat{y}] + r[2\hat{x}+1][2\hat{y}] + r[2\hat{x}][2\hat{y}+1] + r[2\hat{x}+1][2\hat{y}+1]) \quad (124)$$

This is basically same as the brilliant 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  $x, y = 0, 1, \dots, L-1$  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

$$T_c[x][y] = \frac{1}{4} (T_c[x+1][y] + T_c[x-1][y] + T_c[x][y+1] + T_c[x][y-1]) + b_0[x][y] \quad (125)$$

with loops over  $x, y = 0, 1, \dots, L-1$ . 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 Suggested Project Topic & References

One possible Project Topic:

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

[https://en.wikipedia.org/wiki/Heat\\_equation](https://en.wikipedia.org/wiki/Heat_equation)

The simplest problem can be static (no time dependence) but the local Jacobi iteration does give you the time dependence for free! If you are ambitious you may extend the input heat to random thermal fluctuation. Start simple and see how far you can go.

Here is start for reference: [https://en.wikipedia.org/wiki/Thermal\\_simulations\\_for\\_integrated\\_circuits](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.** Each project should include some use of MPI and/or openMP to speed up the problem and a discussion of performance and accuracy. 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](https://en.wikipedia.org/wiki/Heat_equation)

We are looking at this in 2D. So let think about it. Heat gets “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 temperature 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 raises the temperature  $\rho c_p \dot{T}(x, y, t)$  depending on the specific heat times the density of the material  $\rho(x, y)c_p(x, y)$ . All this result in this equation!

$$\dot{q} = \rho c_p \frac{\partial T(x, y, t)}{\partial t} - \nabla \cdot (k \nabla T(x, y, t)) \quad (126)$$

If we ignore the  $x, y$  dependence of  $k, \rho, c_p$  this is just the equation we are using to solve the  $\mathbf{Ax} = \mathbf{b}$  problem in discrete form!

$$\frac{T[x, y, t + \delta t] - T[x, y, t]}{\delta t} = k \frac{T[x + h, y, t] + T[x - h, y, t] + T[x, y + h, t] + T[x, y - h, t] - 4T[x, y, t]}{h^2} + \dot{q}[x, y, t] \quad (127)$$

where for simplicity we have redefined  $k/\rho c_p \rightarrow k$  and  $\dot{q}/\rho c_p \rightarrow \dot{q}$ . Next multiplying through by  $\delta t$ ,

$$T[x, y, t + \delta t] = (1 - 4k\delta t/h^2)T[x, y, t] + (k\delta t/h^2)(T[x + h, y, t] + T[x - h, y, t] + T[x, y + h, t] + T[x, y - h, t]) + \delta t \dot{q} \quad (128)$$

and defining  $\alpha = 4k\delta t/h^2$ . and

$$T[x, y, t + 1] = (1 - \alpha)T[x, y, t] + \alpha(T[x + h, y, t] + T[x - h, y, t] + T[x, y + h, t] + T[x, y - h, t])/4 + \alpha b(x, y) \quad (129)$$

This is just the Jacobi iteration with an overrelaxation parameter:  $\alpha = 4k\delta t/h^2$ . Small  $\alpha$  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 preferentially in different directions. This is not so hard to do. It just puts different weights ( $w_i(x, y)$ ) 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 should be set to  $w(x, y) = w_E(x, y) + w_N(x, y) + w_W(x, y) + w_S(x, y)$ .) If some one is interested, we can try that too. This is an interesting 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

$$\partial T(x, y, t) / \partial t = 0 \quad (130)$$

Eq. 126 above) . Taking  $k = 1$  and add a “mass” term to stabilize it on a periodic lattice this equation is

$$b = m^2 T(x, y) - \nabla^2 T(x, y) \quad (131)$$

instead of Eq. 126 . The same basic steps now led to the matrix problem,

$$0 = - \frac{T[x+h, y] + T[x-h, y] + T[x, y+h] + T[x, y-h] - 4T[x, y]}{h^2} + m^2 T[x][y] + b[x][y] \quad (132)$$

To transfer to coarser lattice changing  $h \rightarrow 2h \rightarrow 4h \dots$  it is important see the spacing  $h$  explicitly:

$$(4 + h^2 m^2) T[x][y] = T[x+h, y] + T[x-h, y] + T[x, y+h] + T[x, y-h] + b[x, y] \quad (133)$$

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

$$T[x, y] = \text{scale}_h (T[x+h, y] + T[x-h, y] + T[x, y+h] + T[x, y-h]) + \text{scale}_h b[x, y] \quad (134)$$

where

$$\text{scale}_h = 1 / (4 + h^2 m^2) \quad (135)$$

**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

$$\text{scale}(lev) = 1 / (2 + 2^{2lev} h^2 m^2) \quad (136)$$

for levels:  $lev = 0, 1, \dots$ . (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

$$\phi[x] = \text{scale}(0) * (\phi[x+1] + \phi[x-1]) + res[x] \quad (137)$$

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

$$res[x] = \text{Project}(res[x]) \quad (138)$$

and iterate:

$$\phi[x] = \text{scale}(lev) * (\phi[x+1] + \phi[x-1]) + res[x] \quad (139)$$

Going back up (‘coarse to fine’) we combined “interpolate plus add” routines,

$$\phi_{fine}[x] = \phi_{fine}[x] + \text{Interpolate}(\phi_{coarse}[x]) \quad (140)$$

Here `phi_{coarse}[x]` is the error contributing 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 interpolate 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 algorithms for the Laplace equation or (2) Time dependence of the Heat equation. 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 of 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 solve the Laplace equation  $\nabla^2\phi = \dot{q}$  or convert to it as the end point of the thermal diffusion  $\dot{T} = -\nabla^2T - \dot{q} \rightarrow 0$ . These are linear equations of the elliptical ( $x$ ) and parabolic ( $x$ ) form. There is one more important example the hyperbolic or wave equation,

$$\frac{\partial^2\phi}{\partial t^2} = \nabla^2\phi + f(x, t) \quad (141)$$

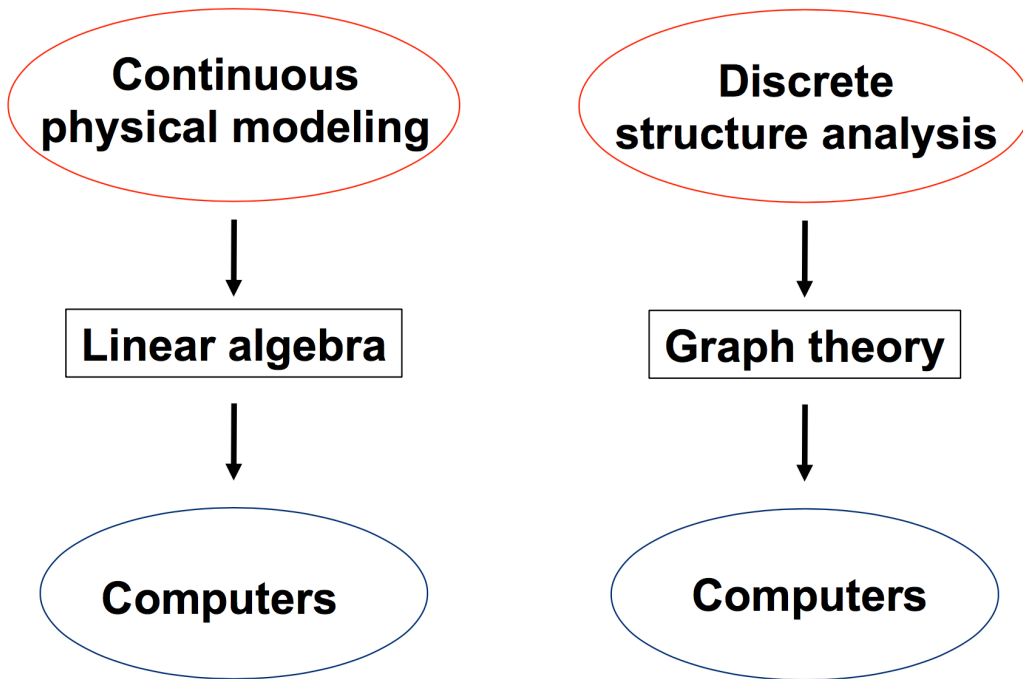
with or without a driving term. This gives you waves (radio, light and now even gravity!). Harder because it oscillates like our pendulum problem! (By the way with an  $i$  in front of the diffusion (or parabolic equation) it becomes the Schrodinger wave equation! The factors of  $i$  are all important as we saw 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 slides John Gilbert with pretty pictures to give you an overview.

The second important theme is that both PDEs and Linear Algebra come from minimization of quadratic (forms). Finally FEM says start with continuum, approximate 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 hits on it!)



Here is a brief idea of the relation.

### 12.1 From Matrices to Graphs to Geometry

A Square matrix 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 introplationg and the demystification of FEM

See FEM page at (where else) Wikipedia:

[https://en.wikipedia.org/wiki/Finite\\_element\\_method](https://en.wikipedia.org/wiki/Finite_element_method)