LECTURE 2
What is the Fast Multipole Method?

• An algorithm for achieving fast products of particular dense matrices with vectors
• Similar to the Fast Fourier Transform
  – For the FFT, matrix entries are uniformly sampled complex exponentials
• For FMM, matrix entries are
  – Derived from particular functions
  – Functions satisfy known “translation” theorems
• Name is a bit unfortunate
  – What the heck is a multipole? We will return to this …
• Why is this important?
Outline

• Revision of the last class
  – Matrices
  – Complexity
  – Manipulation of series
  – Order notation

• Integral Equations
  – Separable/Degenerate Kernels

• Fast Fourier Transform
  – Danielson Lanczos Lemma
  – Bit reversal
  – Remarks

• Homework
Vectors and Matrices

\(d\) dimensional column vector \(\mathbf{x}\) and its transpose

\[
\mathbf{x} = \begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_d
\end{pmatrix}
\quad \text{and} \quad
\mathbf{x}^T = (x_1 \ x_2 \ \ldots \ x_d)
\]

- \(n \times d\) dimensional matrix \(\mathbf{M}\) and its transpose \(\mathbf{M}^T\)

\[
\mathbf{M} = \begin{pmatrix}
m_{11} & m_{12} & m_{13} & \ldots & m_{1d} \\
m_{21} & m_{22} & m_{23} & \ldots & m_{2d} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
m_{n1} & m_{n2} & m_{n3} & \ldots & m_{nd}
\end{pmatrix}
\quad \text{and}
\]

\[
\mathbf{M}^T = \begin{pmatrix}
m_{11} & m_{12} & \ldots & m_{1d} \\
m_{12} & m_{22} & \ldots & m_{2d} \\
m_{13} & m_{23} & \ldots & m_{3d} \\
\vdots & \vdots & \ddots & \vdots \\
m_{1d} & m_{2d} & \ldots & m_{nd}
\end{pmatrix}
\]
Matrix vector product

\[ s_1 = m_{11} x_1 + m_{12} x_2 + \ldots + m_{1d} x_d \]
\[ s_2 = m_{21} x_1 + m_{22} x_2 + \ldots + m_{2d} x_d \]
\[ \ldots \]
\[ s_n = m_{n1} x_1 + m_{n2} x_2 + \ldots + m_{nd} x_d \]

- Matrix vector product is identical to a sum
  \[ s_i = \sum_{j=1}^{d} m_{ij} x_j \]
- So algorithm for fast matrix vector products is also a fast summation algorithm
- \( d \) products and sums per line
- \( N \) lines
- Total \( Nd \) products and \( Nd \) sums to calculate \( N \) entries
Linear Systems

- Solve a system of equations
  \[ Mx = s \]
  
- \( M \) is a \( N \times N \) matrix, \( x \) is a \( N \) vector, \( s \) is a \( N \) vector

- Direct solution (Gauss elimination, LU Decomposition, SVD, …) all need \( O(N^3) \) operations

- Iterative methods typically converge in \( k \) steps with each step needing a matrix vector multiply \( O(N^2) \)
  - if properly designed, \( k \ll N \)

- A fast matrix vector multiplication algorithm (\( O(N \log N) \) operations) will speed all these algorithms
Memory complexity

- Sometimes we are not able to fit a problem in available memory
  - Don’t care how long solution takes, just if we can solve it
- To store a $N \times N$ matrix we need $N^2$ locations
  - 1 GB RAM = $1024^3 = 1,073,741,824$ bytes
  - => largest $N$ is 32,768
- “Out of core” algorithms copy partial results to disk, and keep only necessary part of the matrix in memory
- FMM allows reduction of memory complexity as well
  - *Elements of the matrix required for the product can be generated as needed*
Dense and Sparse matrices

• Operation estimates are for dense matrices.
  – Majority of elements of the matrix are non-zero

• However in many applications matrices are sparse

• A sparse matrix (or vector, or array) is one in which most of the elements are zero.
  – Can store a sparse matrix as a list of (index, value) pairs, and save on storage space.
  – For a given sparsity structure it is possible to define a fast matrix-vector product algorithm
  – May be possible to also solve the linear system cheaply
Structured matrices

• Fast algorithms have been found for many dense matrices
• Typically the matrices have some “structure”
• Definition:
  – A dense matrix of order $N \times N$ is called structured if its entries depend on only $O(N)$ parameters.
• Most famous example – the fast Fourier transform
Is the FMM a structured matrix algorithm?

• FFT and other algorithms work on structured matrices
• What about FMM?
• Speeds up matrix-vector products (sums) of a particular type

\[
s(y_j) = \sum_{i=1}^{N} a_i \phi(x_i, y_j)
\]

\[
s = \Phi a
\]

• Above sum also depends on \(O(N)\) parameters \(\{x_i\}, \{y_j\}, \phi\)
• FMM can be thought of as working on “loosely” structured matrices
A very simple algorithm

• Not FMM, but has some key ideas
• Consider
  \[ S(x_i) = \sum_{j=1}^{N} \alpha_j (x_i - y_j)^2 \quad i=1, \ldots, M \]
• Naïve way to evaluate the sum will require \( MN \) operations
• Instead can write the sum as
  \[ S(x_i) = (\sum_{j=1}^{N} \alpha_j) x_i^2 + \left( \sum_{j=1}^{N} \alpha_j y_j^2 \right) - 2x_i \left( \sum_{j=1}^{N} \alpha_j y_j \right) \]
  – Can evaluate each bracketed sum over \( j \) and evaluate an expression of the type
  \[ S(x_i) = \beta x_i^2 + \gamma - 2x_i \delta \]
  – Requires \( O(M+N) \) operations
• Key idea – use of analytical manipulation of series to achieve faster summation
Integral Equation

• FMM is often used in integral equations

• What is an integral equation?

\[ \int k(x, y)u(x)dx + au(y) = f(y) \]

\[ \int k(x, y)u(x)dx = f(y) \]

• Function \( k(x, y) \) is called the kernel

• Integral equations are typically solved by “quadrature”
  – Quadrature is the process of approximately evaluating an integral

• If we can write

\[ \int k(x, y)u(x)dx = \sum_{j=1}^{N} k(x_j, y)u(x_j)w_j \]
Integral Equations

- Equation can be written as
  \[ \sum_{j=1}^{N} k(x_j, y_i) w_j u(x_j) + a u(y_i) = f(y_i), \quad i = 1, \ldots, N \]

- Becomes a matrix vector equation

  \[
  \begin{bmatrix}
  w_1 k_{11} + a & w_2 k_{12} & w_3 k_{13} & w_4 k_{14} \\
  w_1 k_{21} & w_2 k_{22} + a & w_3 k_{23} & w_4 k_{24} \\
  w_1 k_{31} & w_2 k_{32} & w_3 k_{33} + a & w_4 k_{34} \\
  w_1 k_{41} & w_2 k_{42} & w_3 k_{43} & w_4 k_{44} + a \\
  \end{bmatrix}
  \begin{bmatrix}
  u_1 \\
  u_2 \\
  u_3 \\
  u_4 \\
  \end{bmatrix}
  =
  \begin{bmatrix}
  f_1 \\
  f_2 \\
  f_3 \\
  f_4 \\
  \end{bmatrix}
  \]

- If kernel is degenerate (or separable)

  \[ k(x, y) = \sum_{l=1}^{p} \psi_l(x) \phi_l(y) \]
\[ k(x, y) = \sum_{l=1}^{p} \psi_l(x) \phi_l(y) \]

\[ \sum_{j=1}^{N} k(x_j, y_i) w_j u(x_j) + au(y_i) = f(y_i), \quad i = 1, \ldots, N \]
Approximate evaluation

• FMM introduces another key idea or “philosophy”
  – In scientific computing we almost never seek exact answers
  – At best, “exact” means to “machine precision”

• So instead of solving the problem we can solve a “nearby” problem that gives “almost” the same answer

• If this “nearby” problem is much easier to solve, and we can bound the error analytically we are done.

• In the case of the FMM
  – Manipulate series to achieve approximate evaluation
  – Use analytical expression to bound the error

• FFT is exact … FMM can be arbitrarily accurate
Applications – I Interpolation

• Given a scattered data set with points and values \( \{x_i, f_i\} \)
• Build a representation of the function \( f(x) \)
  – That satisfies \( f(x_i) = f_i \)
  – Can be evaluated at new points
• One approach use “radial-basis functions”
  \[
  f(x) = \sum_{i=1}^{N} \alpha_i R(x - x_i) + p(x) \\
  f_j = \sum_{i=1}^{N} \alpha_i R(x_j - x_i) + p(x_j)
  \]
• Two problems
  – Determining \( \alpha_i \)
  – Knowing \( \alpha_i \) determine the product at many new points \( x_j \)
• Both can be solved via FMM (Cherrie et al, 2001)
Applications 2

- RBF interpolation

Cherrie et al 2001
Applications 3

• Sound scattering off rooms and bodies
  – Need to know the scattering properties of the head and body (our interest)

\[ \nabla^2 P + k^2 P = 0 \]

\[ \frac{\partial P}{\partial n} + i\sigma P = g \]

\[ \lim_{r \to \infty} \left( \frac{\partial P}{\partial r} - ikP \right) = 0 \]

\[ C(x)p(x) = \int_{\Gamma_y} \left[ G(x, y; k) \frac{\partial}{\partial n_y} p(y) - \frac{\partial}{\partial n_y} G(x, y; k) p(y) \right] d\Gamma_y \]

\[ G(x, y) = \frac{e^{ik|x-y|}}{4\pi|x-y|} \]
EM wave scattering

- Similar to acoustic scattering
- Send waves and measure scattered waves
- Attempt to figure out object from the measured waves
- Need to know “Radar cross-section”
- Many applications
  - Light scattering
  - Radar
  - Antenna design
  - ....
Molecular and stellar dynamics

- Many particles distributed in space
- Particles exert a force on each other
- Simplest case force obeys an inverse-square law (gravity, coulombic interaction)

\[
\frac{d^2 \mathbf{x}_i}{dt^2} = F_i, \quad F_i = \sum_{\substack{j=1 \atop j \neq i}}^{N} q_i q_j \frac{(\mathbf{x}_i - \mathbf{x}_j)}{|\mathbf{x}_i - \mathbf{x}_j|^3}
\]

Figure 10: Slice views of the 5CB cluster at time 0 and 1.16 ns. The slices are passing the spheric center with thickness of 70 Å.
Complexity

• The most common complexities are
  – $O(1)$ - not proportional to any variable number, i.e. a fixed/constant amount of time
  – $O(N)$ - proportional to the size of $N$ (this includes a loop to $N$ and loops to constant multiples of $N$ such as $0.5N$, $2N$, $2000N$ - no matter what that is, if you double $N$ you expect (on average) the program to take twice as long)
  – $O(N^2)$ - proportional to $N$ squared (you double $N$, you expect it to take four times longer - usually two nested loops both dependent on $N$).
  – $O(\log N)$ - this is trickier to show - usually the result of binary splitting.
  – $O(N \log N)$ this is usually caused by doing $\log N$ splits but also doing $N$ amount of work at each "layer" of splitting.
Big Oh

• Equivalently,

  \[ f(n) = O(g(n)) \]

  \[ \exists c, n_0 \geq 0 \ \forall n \geq n_0 \ |f(n)| \leq c \cdot g(n) \]
Log complexity

• If you half data at each stage then number of stages until you have a single item is given (roughly) by $\log_2 N$. => binary search takes $\log_2 N$ time to find an item.
• All logs grow a constant amount apart (homework)
  – So we normally just say $\log N$ not $\log_2 N$.
• Log $N$ grows very slowly
Fast Fourier Transform

• Fourier transform of a function $h(t)$ is given by $H(f)$ where $f$ is the frequency

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i ft} dt$$

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi i ft} df$$

• Discrete Fourier Transform: if the function is sampled at discrete times

$$h_k \equiv h(t_k), \quad t_k \equiv k\Delta, \quad k = 0, 1, 2, \ldots, N - 1$$

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt \approx \sum_{k=0}^{N-1} h_k e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N}$$
Discrete Fourier Transform

• All notion of time has disappeared
• Multiplication of sampled data by a matrix

\[
H_n \equiv \sum_{k=0}^{N-1} h_k e^{2\pi i kn/N}
\]

• This matrix is called the Fourier Matrix
• As discussed earlier it is a structured matrix
A Fourier matrix of order $n$ is defined as the following

$$
\begin{bmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & W & W^2 & \cdots & W^{n-1} \\
1 & W^2 & W^4 & \cdots & W^{2(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & W^{n-1} & W^{2(n-1)} & \cdots & W^{(n-1)(n-1)}
\end{bmatrix},
$$

where

$$W = e^{\frac{2\pi i}{n}},$$

is an $n$th root of unity.
Fast Fourier Transform

- Presented by Cooley and Tukey in 1965, but invented several times, including by Gauss (1809) and Danielson & Lanczos (1948)
- Danielson Lanczos lemma

\[
F_k = \sum_{j=0}^{N-1} e^{2\pi i j k / N} f_j
\]

\[
= \sum_{j=0}^{N/2-1} e^{2\pi i k (2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi i k (2j+1)/N} f_{2j+1}
\]

\[
= \sum_{j=0}^{N/2-1} e^{2\pi i k j / (N/2)} f_{2j} + e^{2\pi i k / (N/2)} W^k \sum_{j=0}^{N/2-1} e^{2\pi i k j / (N/2)} f_{2j+1}
\]

\[
= F_k^e + W^k F_k^o
\]
FFT

- So DFT of order $N$ can be expressed as sum of two DFTs of order $N/2$
- Does this improve the complexity?
- Yes \((N/2)^2 + (N/2)^2 = \frac{N^2}{2} < N^2\)
- But we are not done ….
- Can apply the lemma recursively
  \[
  F_k^e = F_k^{ee} + W^k F_k^{eo}, \quad F_k^o = F_k^{oe} + W^k F_k^{oo},
  \]
- Finally we have a set of one point transforms
- One point transform is identity \(F_k^{eeeeoo...oe} = f_n\)
Complexity

- Each $F_k$ is a sum of $\log_2 N$ transforms and (factors)
- There are $N$ $F_k$'s
- So the algorithm is $O(N \log_2 N)$
FFT and bit-shifts

• Set $o$ to 1 and $e$ to 0
• Then the sequence $ooeoeo...$ can be interpreted as a binary number
• Reversing the pattern of $e$’s and $o$’s gives the binary value of $n$