Fast Multipole Methods: Fundamentals & Applications

Ramani Duraiswami
Nail A. Gumerov

Matrix vector product

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

- 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

A very simple algorithm

- Not FMM, but has some key ideas
- Consider

$$S(x_i)=\sum_{j=1}^{N} \alpha_j (x_i - y_j)^2 \qquad i=1, \ldots ,M$$

- Naïve way to evaluate the sum will require MN operations
- Instead can write the sum as

$$S(x_i)=(\sum_{j=1}^{N} \alpha_j)x_i^2 + (\sum_{j=1}^{N} \alpha_j y_j^2) -2x_i(\sum_{j=1}^{N} \alpha_j y_j)$$

  - 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
- Matrix is structured … determined by $N+M$ quantities

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

Fast Multipole Methods (FMM)

- Introduced by Rokhlin & Greengard in 1987
- Called one of the 10 most significant advances in computing of the 20th century
- Speeds up matrix-vector products (sums) of a particular type

$$s(x_j) = \sum_{i=1}^{N} \alpha_i \phi(x_j - x_i), \quad \{s_j\} = [\Phi_{ji}]\{\alpha_i\}.$$

- Above sum requires $O(MN)$ operations.
- For a given precision ε the FMM achieves the evaluation in $O(M+N)$ operations.

Reduction of Complexity

Straightforward (nested loops):

$$
\begin{aligned}
&\text{for } j = 1,\ldots,M \\
&\quad v_j = 0; \\
&\quad \text{for } i = 1,\ldots,N \\
&\quad\quad v_j = v_j + \Phi(\mathbf{y}_j, \mathbf{x}_i) u_i; \\
&\quad \text{end;} \\
&\text{end;}
\end{aligned}
$$

Complexity: $O(MN)$

Factorize matrix entries

$$\Phi(y_j, x_i) = \sum_{m=0}^{p-1} a_m(\mathbf{x}_i - \mathbf{x}_*) f_m(\mathbf{y}_j - \mathbf{x}_*)$$

If $p \ll \min(M,N)$ then complexity reduces!

Factorized:

$$
\begin{aligned}
&\text{for } m = 0,\ldots,p-1 \\
&\quad c_m = 0; \\
&\quad \text{for } i = 1,\ldots,N \\
&\quad\quad c_m = c_m + a_m(\mathbf{x}_i - \mathbf{x}_*) u_i; \\
&\quad \text{end;} \\
&\text{end;}
\end{aligned}
$$

$$
\begin{aligned}
&\text{for } j = 1,\ldots,M \\
&\quad v_j = 0; \\
&\quad \text{for } m = 0,\ldots,p-1 \\
&\quad\quad v_j = v_j + c_m f_m(\mathbf{y}_j - \mathbf{x}_*); \\
&\quad \text{end;} \\
&\text{end;}
\end{aligned}
$$

Complexity: $O(pN+pM)$

Approximate evaluation

- FMM introduces another key idea or "philosophy"
  - In scientific computing we almost never get 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

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

Fast Fourier Transforms

- Allow matrix vector product (Fourier transform), and linear system solution (inverse Fourier transform) of data of size N in N log N time
- Require the data to be uniformly sampled

DFT and its inverse for periodic discrete data

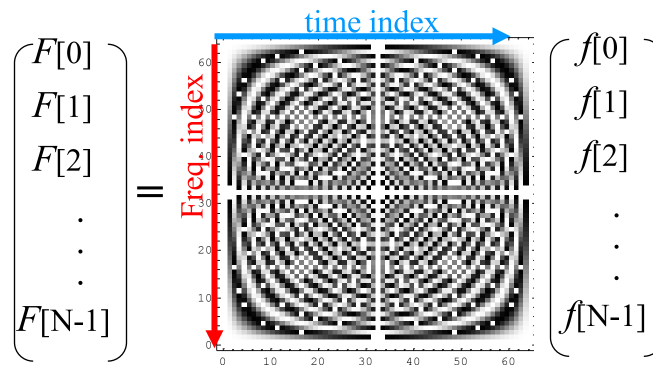
$$\Phi[k] = \sum_{n=0}^{N-1} \phi[n]\ e^{-2\pi i k n h/p} \qquad p = N h$$

$$= \sum_{n=0}^{N-1} \phi[n]\ e^{-2\pi i k n/N}$$

Discrete time Numerical Fourier Analysis

DFT is really just a matrix multiplication!

$$F[m] = 1/N \sum_{k=0}^{N-1} e^{-2 \pi i k m/N} f[k]$$



$$F = F_N \; f$$

Fourier Matrices

A Fourier matrix of order n is defined as the following

$$F_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix},$$

where

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

is an nth root of unity.

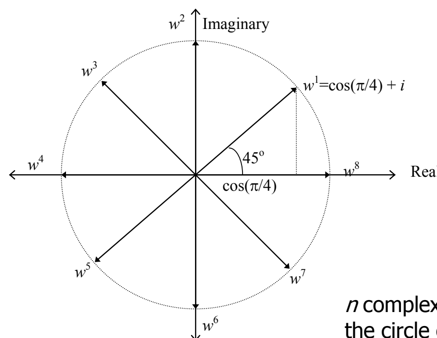
$i=\sqrt{-1}$ Primitive Roots of Unity

A number ω is a ***primitive n-th root of unity***, for n>1, if

$\omega^n = 1$

The numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are all distinct

- Example: The complex number $e^{2\pi i/n}$ is a primitive n-th root of unity, where



properties

1. $\omega^1 = e^{\frac{2\pi i}{n}} \neq 1$
2. $\omega^n = \left(e^{\frac{2\pi i}{n}}\right)^n = e^{2\pi i} = \cos 2\pi + i \sin 2\pi = 1$
3. $S = \sum_{p=0}^{n-1} \omega^{jp} = \omega^0 + \omega^j + \omega^{2j} + \omega^{3j} + \ldots\ldots + \omega^{j(n-1)} = 0$

n complex roots of unity equally spaced around the circle of unit radius centered at the origin of the complex plane.

Roots of Unity: Properties

- Property 1: Let ω be the principal n^{th} root of unity. If n > 0, then $\omega^{n/2} = -1$.
  - Proof: $\omega = e^{2\pi i/n} \Rightarrow \omega^{n/2} = e^{\pi i} = -1$. (Euler's formula)
  - **Reflective Property:**
  - Corollary: $\omega^{k+n/2} = -\omega^k$.

- Property 2: Let n > 0 be even, and let ω and ν be the principal n^{th} and $(n/2)^{th}$ roots of unity. Then $(\omega^k)^2 = \nu^k$.
  - Proof: $(\omega^k)^2 = e^{(2k)2\pi i/n} = e^{(k)2\pi i/(n/2)} = \nu^k$.
  - **Reduction Property:** If ω is a primitive (2n)-th root of unity, then ω^2 is a primitive n-th root of unity.

- L3: Let $n > 0$ be even. Then, the squares of the n complex n^{th} roots of unity are the $n/2$ complex $(n/2)^{th}$ roots of unity.
  - Proof: If we square all of the nth roots of unity, then each $(n/2)^{th}$ root is obtained exactly twice since:
    - L1 $\Rightarrow$ $\omega^{k+n/2} = -\omega^k$
    - thus, $(\omega^{k+n/2})^2 = (\omega^k)^2$
    - L2 $\Rightarrow$ both of these = ν^k
    - $\omega^{k+n/2}$ and ω^k have the same square
- **Inverse Property:** If ω is a primitive root of unity, then $\omega^{-1}=\omega^{n-1}$
  - Proof: $\omega\omega^{n-1}=\omega^n=1$

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

$$
\begin{aligned}
F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\
&= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\
&= F_k^e + W^k F_k^o
\end{aligned}
$$

- So far we have seen what happens on the right hand side
- How about the left hand side?
- When we split the sums in two we have two sets of sums with *N/2* quantities for *N* points.
- So the complexity is $N^2/2 + N^2/2 = N^2$
- So there is no improvement
- Need to reduce the sums on the right hand side as well
  - We need to reduce the number of sums computed from 2N to a lower number
  - Notice that the transforms F_e^k and F_o^k are periodic in *k* with length *N*/2.
  - So we need only compute half of them!

FFT

- So DFT of order *N* can be expressed as sum of two DFTs of order *N/2 evaluated at N/2* points
- Does this improve the complexity?
- Yes $(N/2)^2+(N/2)^2 = N^2/2 < N^2$
- 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 $F_k^{eoeeoeo\cdots oee} = f_n$

Complexity

- Each F_k is a sum of $\log_2$ N transforms and (factors)
- There are N F_k s
- So the algorithm is $O(N \log_2 N)$
- *This is a recursive algorithm*

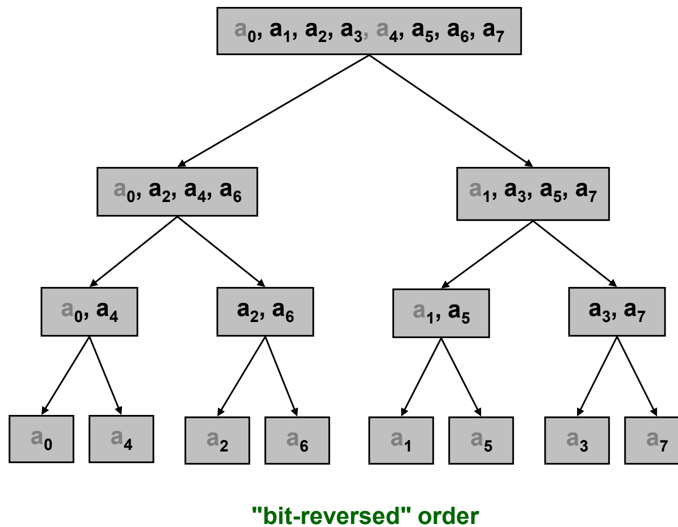
Matlab FFTtx (from Moler)

```
function y = ffttx(x)
%   FFTTX(X) computes the same finite Fourier transform as FFT(X).
%   The code uses a recursive divide and conquer algorithm for
%   even order and matrix-vector multiplication for odd order.
%   If length(X) is m*p where m is odd and p is a power of 2, the
%   computational complexity of this approach is O(m^2)*O(p*log2(p)).

x = x(:);
n = length(x);
omega = exp(-2*pi*i/n);
```

```
if rem(n,2) == 0
   % Recursive divide and conquer
   k = (0:n/2-1)';
   w = omega .^ k;
   u = ffttx(x(1:2:n-1));
   v = w.*ffttx(x(2:2:n));
   y = [u+v; u-v];
else
   % The Fourier matrix.
   j = 0:n-1;
   k = j';
   F = omega .^ (k*j);
   y = F*x;
end
```

Scrambled Output of the FFT



FFT and IFFT

The *discrete Fourier transform* of a vector x is the product $F_n x$.
The *inverse discrete Fourier transform* of a vector x is the product $F_n^* x$.

Both products can be done efficiently using the fast Fourier transform (FFT) and the inverse fast Fourier transform (IFFT) in $O(n \log n)$ time.

The FFT has revolutionized many applications by reducing the complexity by a factor of almost n

Can relate many other matrices to the Fourier Matrix

Circulant Matrices

$$C_n = C(x_1, ..., x_n) = \begin{bmatrix} x_1 & x_n & x_{n-1} & \cdots & x_2 \\ x_2 & x_1 & x_n & \cdots & x_3 \\ x_3 & x_2 & x_1 & \cdots & x_4 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_n & x_{n-1} & x_{n-2} & \cdots & x_1 \end{bmatrix}$$

Toeplitz Matrices

$$T_n = T(x_{-n+1}, \cdots, x_0, ..., x_{n-1}) = \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ x_{-1} & x_0 & x_1 & \cdots & x_{n-2} \\ x_{-2} & x_{-1} & x_0 & \cdots & x_{n-3} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \end{bmatrix}$$

Hankel Matrices

$$H_n = H(x_{-n+1}, \cdots, x_0, \cdots, x_{n-1}) = \begin{bmatrix} x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \\ x_{-n+2} & x_{-n+3} & x_{-n+4} & \cdots & x_1 \\ x_{-n+3} & x_{-n+4} & x_{-n+5} & \cdots & x_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_0 & x_1 & x_2 & \cdots & x_{n-1} \end{bmatrix}$$

Vandermonde Matrices

$$V = V(x_0, x_1, ..., x_n) = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_{n-1} \\ \cdots & \cdots & \cdots & \cdots \\ x_0^{n-1} & x_1^{n-1} & \cdots & x_{n-1}^{n-1} \end{bmatrix}$$

- Modern signal processing very strongly based on the FFT
- One of the defining inventions of the 20th century

Fourier Matrices

A Fourier matrix of order n is defined as the following

$$F_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix},$$

where

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

is an nth root of unity.

FFT and IFFT

The *discrete Fourier transform* of a vector x is the product $F_n x$.
The *inverse discrete Fourier transform* of a vector x is the product $F_n^* x$.

Both products can be done efficiently using the fast Fourier transform (FFT) and the inverse fast Fourier transform (IFFT) in $O(n \log n)$ time.

The FFT has revolutionized many applications by reducing the complexity by a factor of almost *n*

Can relate many other matrices to the Fourier Matrix

Circulant Matrices

$$C_n = C(x_1, ..., x_n) = \begin{bmatrix} x_1 & x_n & x_{n-1} & \cdots & x_2 \\ x_2 & x_1 & x_n & \cdots & x_3 \\ x_3 & x_2 & x_1 & \cdots & x_4 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_n & x_{n-1} & x_{n-2} & \cdots & x_1 \end{bmatrix}$$

Toeplitz Matrices

$$T_n = T(x_{-n+1}, \cdots, x_0, ..., x_{n-1}) = \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ x_{-1} & x_0 & x_1 & \cdots & x_{n-2} \\ x_{-2} & x_{-1} & x_0 & \cdots & x_{n-3} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \end{bmatrix}$$

Hankel Matrices

$$H_n = H(x_{-n+1}, \cdots, x_0, \cdots, x_{n-1}) = \begin{bmatrix} x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \\ x_{-n+2} & x_{-n+3} & x_{-n+4} & \cdots & x_1 \\ x_{-n+3} & x_{-n+4} & x_{-n+5} & \cdots & x_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_0 & x_1 & x_2 & \cdots & x_{n-1} \end{bmatrix}$$

Vandermonde Matrices

$$V = V(x_0, x_1, ..., x_n) = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_{n-1} \\ \cdots & \cdots & \cdots & \cdots \\ x_0^{n-1} & x_1^{n-1} & \cdots & x_{n-1}^{n-1} \end{bmatrix}$$

- Modern signal processing very strongly based on the FFT
- One of the defining inventions of the 20th century

Asymptotic Equivalence

- $f(n) \sim g(n)$

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = 1$$

Little Oh

•*Asymptotically smaller*:

•$f(n) = o(g(n))$

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = 0$$

Big Oh

- *Asymptotic Order of Growth:*
  - $f(n) = O(g(n))$

$$\limsup_{n\to\infty} \left(\frac{f(n)}{g(n)} \right) < \infty$$

The Oh's

$$\text{If } f = o(g) \text{ or } f \sim g \text{ then } f = O(g)$$

$\lim = 0$ $\quad \lim = 1$ $\quad \lim < \infty$

The Oh's

If $f = o(g)$, then $g \neq O(f)$

$$\lim \frac{f}{g} = 0 \qquad \lim \frac{g}{f} = \infty$$

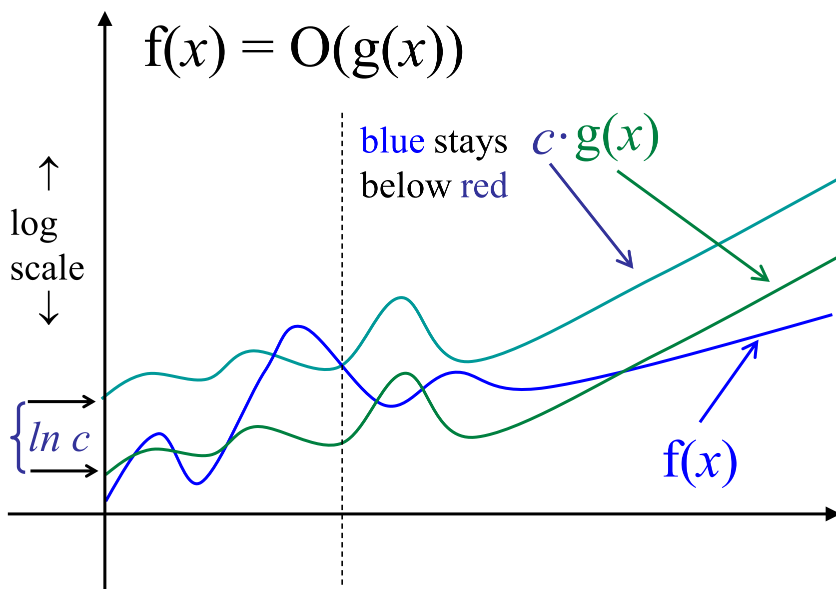
Big Oh

- Equivalently,

- $f(n) = O(g(n))$

$$\exists c, n_0 \geq 0 \ \forall n \geq n_0 \ |f(n)| \leq c \cdot g(n)$$

Big Oh



Separable (Degenerate) Kernels

Compute matrix-vector product

$$\mathbf{v} = \mathbf{Au},$$

or sums

$$v_j = \sum_{i=1}^{N} u_i A(x_i, y_j), \quad j = 1, \ldots, M.$$

Fast computation in case of degenerate (separable) kernel:

$$A(x_i, y_j) = \sum_{m=1}^{n} \varphi_m(x_i) \psi_m(y_j)$$

$$v_j = \sum_{i=1}^{N} u_i \sum_{m=1}^{n} \varphi_m(x_i) \psi_m(y_j) = \sum_{m=1}^{n} \psi_m(y_j) \sum_{i=1}^{N} u_i \varphi_m(x_i) = \sum_{m=1}^{n} c_m \psi_m(y_j),$$

where

$$c_m = \sum_{i=1}^{N} u_i \varphi_m(x_i).$$

Authors: Unknown.