Fast Matrix Algorithms

S. Chandrasekaran Students of 20{18,19,22}

February 28, 2022

January 2, 2018

Definition 1. *A **matrix** is a rectangular array of numbers (usually either real or complex).*

The set of all real matrices with m rows and n columns of real numbers will be denoted as $\mathbb{R}^{m\times n}$. Similarly $\mathbb{C}^{m\times n}$ for complex matrices. We will use capital Roman letters to denote matrices. Small Roman letters will denote matrices with only one column. Matrices with one row will be denoted by taking the transpose of matrices with one column, where the transpose is denoted by superscript T and is defined as

$$A^T_{i,j} = A_{j,i}.$$

1 Matrix arithmetic

Matrix addition, $A = B + C$, is defined as

$$A_{i,j} = B_{i,j} + C_{i,j},$$

which costs mn additions. We will refer to this loosely as $O(mn)$ flops and ignore the cost of memory access (not because it is unimportant).

Scalar multiplication $A = \alpha B$ is defined as

$$A_{i,j} = \alpha B_{i,j}$$

and takes mn multiplications.

For $B \in \mathbb{R}^{m\times k}$ and $C \in \mathbb{R}^{k\times n}$, matrix multiplication $A = BC$ is defined as

$$A_{i,j} = \sum_{l=1}^{k} B_{i,l} C_{l,j}$$

and takes mkn multiplications and $m(k-1)n$ additions, which is loosely denotes as $O(mkn)$ flops. If the matrices are square then we have $O(n^3)$ flops.

Strassen suprised the world by showing that you can decrease the cost substantially.

2 Fast matrix multiplication

We give a brief review of the ideas behind Strassen's algorithm. Consider $AB{=}C$ as a **block**-matrix product

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

with

$$C_{ij} = A_{i1}\, B_{1j} + A_{i2}\, B_{2j}.$$

We know that multiplying two $n \times n$ matrices by the naive method costs $2n^3$ flops while adding two such matrices (or scalar multiplication) only costs n^2 flops. So the idea is to trade block matrix multiplications (which is 8 for the naive method) for block matrix additions (which is 4 for the naive method). Towards that end consider the following template for the algorithm

$$M_l = \left(\sum_{i,j=1}^{2} \alpha_{ij}^l A_{ij} \right) \left(\sum_{i,j=1}^{2} \beta_{ij}^l B_{ij} \right), \qquad l = 1, \ldots, L,$$

and

$$C_{ij} = \sum_{l=1}^{L} \lambda_{ij}^l M_l, \qquad i, j = 1, 2.$$

The goal is to find scalars α_{ij}^l, β_{ij}^l and λ_{ij}^l such that C_{ij} is correct. $L = 7$ is the smallest decrease one can hope for and the one that Strassen was successful with.

We see that we have

$$L \times (8 + 4) = 12L$$

unknown scalars in total, and we have

$$16 \times 4 = 64$$

equations in total when we match $A_{ij} B_{pq}$ on both sides of the equation for C_{ij}. However the system of equations is non-linear so one has to be careful in the search for a solution (they may be inconsistent for example). Strassen found one using just ± 1 and 0 for $L = 7$.

Note that these tri-linear equations are very special (for example, their coefficients are in $\{\pm 1, 0\}$, and they satisfy many symmetries.).

Assuming that the cost to multiply two $n \times n$ matrices in Strassen's method is $f(n)$, we have

$$f(n) \leqslant 7 f(n/2) + cn^2.$$

Then using the estimation techniques in section 3 we obtain that Strassen's method requires $O(n^{\log_2 7})$ flops! Strassen's method is actually useable in practice; the constant hidden in the big-O notation are not too bad and the numerical stability is not an issue. However, the gains can be marginal, and it has not proven popular in practice on modern hardware. It is conjectured that for any $\varepsilon > 0$ there is a fast matrix-matrix multiplication algorithm with asymptotic cost $O(n^{2+\varepsilon})$. The current world record is $\varepsilon > 0.3$, but these algorithms are quite impractical so far and we will say not more about them in these notes.

3 Recursive inequalities

Consider a recursive upper bound of the form

$$f(n) \leqslant c f(n/2) + x(n), \qquad n \in \mathbb{N}.$$

We assume that $f(n) \geqslant 0$, and see that we can expand the above expression, for $n = 2^m$, into

$$f(n) \leqslant c^{m+1} f(1) + \sum_{l=0}^{m} c^l \, x(n/2^l).$$

Now suppose that $x(n) = d n^p \log_2^k n$. Then the sum we need to bound is:

$$\begin{aligned} \sum_{l=0}^{m} c^l \, d \, \frac{n^p}{2^{pl}} \log_2^k \frac{n}{2^l} &= d n^p \sum_{l=0}^{m} \left(\frac{c}{2^p} \right)^l (\log_2 n - l)^k \\ &\leqslant d n^p \log_2^k n \, \frac{\left(\frac{c}{2^p}\right)^{m+1} - 1}{\frac{c}{2^p} - 1}, \qquad c > 2^p, \\ &\leqslant e c^m \log_2^k n. \end{aligned}$$

So the total bound is

$$f(n) \leqslant gc^m \log_2^k n = gn^{\log_2 c} \log_2^k n$$

for some constant g **if** $c > 2^p$. **If** $c = 2^p$ the bound is

$$f(n) \leqslant gn^p \log_2^{k+1} n.$$

If $c < 2^p$ then

$$f(n) \leqslant gn^p \log_2^k n.$$

Aside: Mostly we only need bounds on sums of the form

$$\sum_n n^k \log^p n$$

and these can be upper and lower bounded by relating it to the integral

$$\int x^k \ln^p x \, \mathrm{d}x$$

which does have a closed form expression (for help see *CRC Standard Mathematical Tables and Formulae*, editor Daniel Zwillinger, for example).

4 Bit peeking

Looking at Strassen's method one might wonder if we can further reduce the number of multiplications by looking at the bits representing the numbers of the matrices directly. If the number of bits are finite (which is not assumed in the Strassen case) then one can indeed cheat and produce an $O(n^2)$ "flops" matrix multiplication algorithm. But all that happens in this case is that an extra factor of n goes into the hidden constant. This is not an issue unless one is interested in proving lower bounds for complexities. In these cases one must either work with full Turing complexity, or, work in the real field without bit peeking.

It is easy to see the technique in the case of 2 multiplies, xy and uv. Let's suppose that all of these 4 numbers have finite length base representations: $x = 2^{m_x} x_{m_x} + \cdots + 2^{n_x} x_{n_x}$ for $m_x \geqslant n_x$, etc.. Then we see that xy only occupies bit positions from $m_x + m_y + 1$ to $n_x + n_y$ at most. Therefore it follows that there exists integers a, b, c, d, such that $z = (2^a x + 2^b y)(2^c u + 2^d v)$ has the bit representations of xy and uv in non-overlapping positions inside the product z. Therefore 4 products can be reduced to "1" multiplication. Note that the Turing level complexity has not decreased via this approach.

5 Gaussian elimination

One of the oldest and most used algorithms is Gauss's elimination method. We will consider the equivalent $A = LU$ factorization of a square matrix. This is best described recursively using block matrices as

$$A = \begin{pmatrix} A_1 = L_1 U_1 & B \\ C & D \end{pmatrix} = \begin{pmatrix} L_1 & 0 \\ CU_1^{-1} & I \end{pmatrix} \begin{pmatrix} U_1 & L_1^{-1} B \\ 0 & D - CU_1^{-1} L_1^{-1} B \end{pmatrix},$$

when A_1 has an LU factorization. The sub-matrix

$$S = D - CU_1^{-1} L_1^{-1} B$$

is smaller than A and is called a Schur complement of A and will play a vital role. Assuming that we can compute an LU factorization of $S = L_2 U_2$ we obtain an LU factorization of A

$$A = \begin{pmatrix} L_1 & 0 \\ CU_1^{-1} & L_2 \end{pmatrix} \begin{pmatrix} U_1 & L_1^{-1} B \\ 0 & U_2 \end{pmatrix}.$$

To make this algorithm complete we note that

$$\begin{pmatrix} A & 0 \\ B & C \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ -C^{-1}BA^{-1} & C^{-1} \end{pmatrix},$$

and similarly for (block) upper triangular matrices. This gives a fully recursive recipe for computing the LU factorization of a matrix.

Using the same recursive technique one can estimate that the cost of computing the LU factorization of an $n \times n$ matrix (assuming it exists) is the same as matrix-matrix multiplication! So if we do not use Strassen like techniques the cost is $O(n^3)$ flops.

6 Goals

Our goal in this course is to consider fast algorithms for solving linear equations $Ax = b$, where $A \in \mathbb{R}^{n \times n}$. In particular we will concentrate on direct methods like Gaussian elimination. The goal is to find x in faster than $O(n^3)$ flops, while producing practical algorithms. Algorithms of a purely theoretical bent will not be given a lot of space and time.

Due to the close relationship between solving Gaussian elimination and fast matrix multiplication, we will also consider the latter. Furthermore it is possible to solve equations without appealing to division at all. For example if $\|A\| < 1$ in any sub-multiplicative norm and $(I - A)x = b$, then

$$x = \sum_{n=0}^{\infty} A^n b.$$

In fact there exist coefficients α_k dependent only on A (not b) such that

$$x = \sum_{k=0}^{n} \alpha_k A^k b,$$

when $A \in \mathbb{R}^{n \times n}$. These ideas form the heart of iterative methods for solving $Ax = b$ as it is possible to design fast solvers as long as $A^k b$ and the α_k's can be computed quickly. However iterative methods have their own issues in general, and so we will pay a lot more attention to designing fast *direct* solvers when possible.

Moreover given the practical difficulties of accelerating generic matrix-matrix multiplication, we will turn our attention to families of matrices that have special patterns. Here a lot of progress has been made and they have yielded algorithms of great practical significance.

7 Fourier

The Fourier series has played a significant role in human history (even before Fourier) and there is a matrix that goes along with it. Let $\omega_n = \exp\left(\iota \frac{2\pi}{n}\right)$, where $\iota^2 = -1$ and

$$\exp(z) = \sum_{n=0}^{\infty} \frac{z^n}{n!} = e^z.$$

Let

$$F[n] = \frac{1}{\sqrt{n}} \begin{pmatrix} \omega_n^{0\cdot 0} & \omega_n^{0\cdot 1} & \omega_n^{0\cdot 2} & \cdots & \omega_n^{0\cdot (n-1)} \\ \omega_n^{1\cdot 0} & \omega_n^{1\cdot 1} & \omega_n^{1\cdot 2} & \cdots & \omega_n^{1\cdot (n-1)} \\ \omega_n^{2\cdot 0} & \omega_n^{2\cdot 1} & \omega_n^{2\cdot 2} & \cdots & \omega_n^{2\cdot (n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ \omega_n^{(n-1)\cdot 0} & \omega_n^{(n-1)\cdot 1} & \omega_n^{(n-1)\cdot 2} & \cdots & \omega_n^{(n-1)\cdot (n-1)} \end{pmatrix},$$

where in general

$$F_{i,j}[n] = \omega_n^{ij}, \qquad\qquad i,j = 0,\ldots,n-1.$$

This family of matrices has many interesting properties. Observe that for any i

$$\sum_{j=0}^{n-1} \omega_n^{i\cdot j} = \frac{\omega_n^{i\cdot n}-1}{\omega_n^i - 1} = 0,$$

if $\omega_n^i \neq 1$ and n if $\omega_n^i = 1$.

Also note that

$$\overline{\omega}_n = \exp\left(-\iota\frac{2\pi}{n}\right) = \omega_n^{n-1} \neq 1.$$

From these it follows that

$$F^H[n]\,F[n] = I$$

which shows that F is a unitary matrix and superscript H denotes the Hermitian transpose of a matrix, $A^H_{i,j} = \overline{A}_{j,i}$.

For amusement (exercise) show that

$$F^2[n] = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 1 \\ \vdots & \vdots & \iddots & \iddots & 0 \\ \vdots & 0 & \iddots & & \vdots \\ 0 & 1 & 0 & \cdots & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & E[n-1] \end{pmatrix},$$

where E is called the reverse identity matrix.

From this it follows that

$$F^4[n] = I.$$

From this you can infer the eigenvalues of $F[n]$. Much harder is to find an eigenbasis (see "On the eigenvectors of Schur's matrix," by Patrick Morton in *J. Number Theory*, 1980). For Fourier transforms Hermite polynomials can be chosen as an eigenbasis (same eigenvalues).

One can compute $F[n]x$ in $O(n\log_2 n)$ flops using a technique attributed to Gauss and Cooley & Tukey. We will present this as a block matrix factorization when $n = 2^m$. The general case will be presented later.

Define $\Pi[2m]$, the inverse of the perfect shuffle permutation

$$\Pi[2m]\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{2m-1} \end{pmatrix} = \begin{pmatrix} x_0 \\ x_2 \\ \vdots \\ x_{2m-2} \\ x_1 \\ x_3 \\ \vdots \\ x_{2m-1} \end{pmatrix} = \begin{pmatrix} y_e \\ y_o \end{pmatrix}.$$

Define the diagonal matrix

$$\Omega[m] = \begin{pmatrix} \omega_{2m}^0 & & & \\ & \omega_{2m}^1 & & \\ & & \ddots & \\ & & & \omega_{2m}^{m-1} \end{pmatrix}.$$

Then it turns out that the following block unitary factorization holds:

$$F[2m]\,\Pi^T[2m] = \frac{1}{\sqrt{2}}\begin{pmatrix} I & \Omega[m] \\ I & -\Omega[m] \end{pmatrix}\begin{pmatrix} F[m] & \\ & F[m] \end{pmatrix}.$$

Using this it immediately follows that by exploiting sparsity in the above equation recursively we can do the multiplication $F[2^m]x$ in $O(m2^m)$ flops. We just need to show that the above formula is indeed correct.

First consider the even-numbered columns of $F[2m]$ when $0 \leqslant i < m$:

$$F_{i,2j}[2m] = \frac{1}{\sqrt{2m}}\,\omega_{2m}^{2ij} = \frac{1}{\sqrt{2m}}\exp\left(\iota\frac{2\pi ij}{m}\right) = \frac{1}{\sqrt{2m}}\,\omega_m^{ij} = \frac{1}{\sqrt{2}}\,F_{i,j}[m],$$

and

$$F_{m+i,2j}[2m] = \frac{1}{\sqrt{2m}}\,\omega_{2m}^{(m+i)2j} = \frac{1}{\sqrt{2m}}\exp\left(\iota\frac{2\pi ij}{m}\right) = \frac{1}{\sqrt{2}}\,F_{i,j}[m].$$

This proves the first block column of the identity. Next consider the odd-numbered columns of $F[2m]$ when $0 \leqslant i < m$:

$$F_{i,2j+1}[2m] = \frac{1}{\sqrt{2m}}\,\omega_{2m}^{i(2j+1)} = \frac{1}{\sqrt{2m}}\exp\left(\iota\frac{2\pi i}{2m}\right)\exp\left(\iota\frac{2\pi ij}{m}\right) = \frac{1}{\sqrt{2}}\omega_{2m}^{i}\,F_{i,j}[m],$$

and

$$\begin{aligned} F_{m+i,2j+1}[2m] &= \frac{1}{\sqrt{2m}}\,\omega_{2m}^{(m+i)(2j+1)} = \frac{1}{\sqrt{2m}}\exp(\iota\pi(2j+1))\exp\left(\iota\frac{2\pi i}{2m}\right)\exp\left(\iota\frac{2\pi ij}{m}\right) \\ &= -\frac{1}{\sqrt{2}}\,\omega_{2m}^{i}\,F_{i,j}[m], \end{aligned}$$

which is the second block column of the identity to be proven.

We will refer to this algorithm loosely as the FFT and note that there are many other variants of it.

Since $F^{-1}[n] = F^H[n]$, the above algorithm is easily modified to handle the inverse of the FFT too. Note that this algorithm depends critically on finding a factorization of the matrix where each factor is itself sparse (or at least recursively so). In general we cannot expect our factors to be both sparse and unitary, so we now turn our attention to sparse matrices in general.

8 Sparse matrices

A matrix $A \in \mathbb{R}^{n\times n}$ is said to be sparse if the number of non-zero entries in A is much smaller than n^2. Note that it is trivial to come up with a fast algorithm for Ax if there is a simple way to recall only the non-zero entries in a row of the matrix. In that case the number of multiplications will be exactly the number of non-zeros of A and we have a fast algorithm.

For example suppose A is a tri-diagonal matrix:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & 0 & \cdots & \cdots & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 & \cdots & 0 \\ 0 & a_{3,2} & a_{3,3} & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & a_{n-1,n} \\ 0 & 0 & \cdots & 0 & a_{n,n-1} & a_{n,n} \end{pmatrix}.$$

Then

$$(Ax)_i = \sum_{j=\max(1,i-1)}^{\min(n,i+1)} a_{i,j}\,x_j,$$

and this can be evaluated in at most $O(n)$ flops.

The harder question is how to solve $Ax = b$ quickly when A is sparse. The key idea is to do Gaussian elimination while making sure not to try and eliminate entries of the matrix that are already 0. Note that an entry might start as a 0 but then during the process of Gaussian elimination proceed to fill-in with a non-zero, in which case it might have to be eliminated further in the process. So estimating the complexity can be quite difficult based on the sparsity pattern of the starting matrix A.

As an example consider the above tri-diagonal matrix A. The first step of Gaussian elimination only requires 3 flops irrespective of the size n as there is at most one non-zero in the first column below the $(1,1)$ entry. The new Schur complement is still a tri-diagonal matrix:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & 0 & \cdots & \cdots & 0 \\ 0 & a_{2,2} - \frac{a_{2,1}a_{1,2}}{a_{1,1}} & a_{2,3} & 0 & \cdots & 0 \\ 0 & a_{3,2} & a_{3,3} & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & a_{n-1,n} \\ 0 & 0 & \cdots & 0 & a_{n,n-1} & a_{n,n} \end{pmatrix}.$$

Therefore we can proceed to recursively compute the LU factorization in $O(n)$ flops, and solve $Ax = b$ also in $O(n)$ flops.

Note that in this case all zeros outside the tri-diagonal band are preserved during Gaussian elimination. We say that the matrix has no fill-in.

The amount of fill-in can be highly sensitive to the ordering of the rows and columns of the matrix. For example consider the arrow-head matrix

$$A = \begin{pmatrix} a_{1,1} & 0 & \cdots & 0 & a_{1,n} \\ 0 & a_{2,2} & \ddots & \vdots & a_{2,n} \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & a_{n-1,n-1} & a_{n-1,n} \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n-1} & a_{n,n} \end{pmatrix}.$$

It is easy to check that in this ordering there is no fill-in during Gaussian elimination and we can compute the LU factorization in $O(n)$ flops. However if we reverse the rows and columns and consider a sparsity pattern of the form

$$F = \begin{pmatrix} f_{1,1} & f_{1,2} & \cdots & \cdots & f_{1,n} \\ f_{2,1} & f_{2,2} & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ f_{n,1} & 0 & \cdots & 0 & f_{n,n} \end{pmatrix},$$

then it is easy to check that during Gaussian elimination everything will fill-in and the cost balloons up to $O(n^3)$ flops. Unfortunately finding the optimal ordering of the rows and columns is known to be an NP-complete problem [M.Yannakakis, 1981]. There are lots of good heuristics that can work well sometimes. However it is best for the user to find a good ordering rather than depend on the heuristics.

There are lots of results in this venerable area, but we will deal with sparse matrices as they arise in the course on a case-by-case basis.

9 Shift-invariant structures

Shift-invariance is a very popular engineering tool to manage complexity. Exponential functions behave particularly nicely under shifts:

$$e^{x+a} = e^a\, e^x.$$

Unsurprisingly exponential transforms (Fourier, Laplace, etc.) play a key role in practice. The subject comes along with a family of structured matrices that are related to the Fourier matrix.

9.1 Circulant structure

A $n \times n$ matrix C is said to be **circulant** if

$$C_{ij} = c_{(i-j) \bmod n},$$

for some column vector c. We will sometimes denote this explicitly as

$$C = \mathrm{Circ}[c] = \begin{pmatrix} c_0 & c_{n-1} & \cdots & \cdots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & & & c_2 \\ c_2 & c_1 & c_0 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ c_{n-2} & & & c_1 & c_0 & c_{n-1} \\ c_{n-1} & c_{n-2} & \cdots & c_2 & c_1 & c_0 \end{pmatrix}.$$

The product $\mathrm{Circ}[c]\, x$ is sometimes referred to as the periodic or circular convolution of c and x

$$(c \circledast x)_i = \sum_{k=0}^{n-1} c_{(i-k) \bmod n}\, x_k.$$

Consider the circular shift down matrix

$$Z[n] = \begin{pmatrix} 0 & \cdots & \cdots & 0 & 1 \\ 1 & 0 & & \vdots & 0 \\ 0 & 1 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix} = \mathrm{Circ}[e_1],$$

which is a circulant permutation matrix, and e_1 is a poor short-hand for the second column of $I[n]$, the $n \times n$ identity matrix. Note that

$$\mathrm{Circ}[c] = \sum_{k=0}^{n-1} c_k\, Z^k[n].$$

From this polynomial representation of circulant matrices we see that the key is to understand the eigendecomposition of $Z[n]$. Luckily this is known completely in terms of the discrete Fourier series.

9.1.1 Eigenstructure of $Z[n]$

Let

$$\Lambda[n] = \begin{pmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n-1} \end{pmatrix}.$$

Note that $\Lambda[n]$ is closely related to $\Omega[n/2]$, but they are not the same. Then it turns out that

$$Z[n] = F[n]\, \Lambda[n]\, F^H[n].$$

The verification is essentially an application of equation 7:

$$\begin{aligned}(Z[n])_{i,j} &= \frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{i\cdot k}\,\omega_n^{k}\,\bar{\omega}_n^{k\cdot j}, \qquad 0 \leqslant i,j < n,\\ &= \frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{k(1+i-j)}\end{aligned}$$

which is not 0 (actually 1) iff $\omega_n^{1+i-j} = 1$, which happens iff $1+i-j = qn$ for some integer q. For $q=0$ we have $j = i+1$ as the only solution set, and for $q=1$, the solution set is $i=0$ and $j=n-1$. The constraint $0 \leqslant i,j < n$, prevents any other solutions.

9.1.2 Circulant and FFT

We can see from equation 9.1 that

$$\mathrm{Circ}[c] = F[n]\left(\sum_{k=0}^{n-1} c_k\,\Lambda^k[n]\right)F^H[n] = F[n]\,\mathrm{Diag}[\hat{c}]\,F^H[n]$$

where

$$\mathrm{Diag}[\hat{c}] = \begin{pmatrix} \hat{c}_0 & & & \\ & \hat{c}_1 & & \\ & & \ddots & \\ & & & \hat{c}_{n-1} \end{pmatrix}$$

and

$$\hat{c}_i = \sum_{k=0}^{n-1} c_k \exp\left(\iota\,\frac{2\pi i k}{n}\right).$$

Therefore

$$\hat{c} = \sqrt{n}\,F[n]\,c,$$

and can be computed rapidly with an FFT. Hence $\mathrm{Circ}[c]\,x$ can be computed for the price of 3 FFT's plus $O(n)$ flops when $n = 2^m$.

When that is not the case we can embed the circulant matrix into a larger circulant matrix of size 2^m:

$$\mathrm{Circ}\left[\begin{pmatrix} c \\ * \\ c_{1:n-1} \end{pmatrix}\right]\begin{pmatrix} x \\ 0 \end{pmatrix} = \begin{pmatrix} \mathrm{Circ}[c]\,x \\ * \end{pmatrix}.$$

Note that to be efficient we choose the smallest integer m such that $2n-1 \leqslant 2^m$.

As an example the embedding in the $n=3$ case:

$$\begin{pmatrix} c_0 & c_2 & c_1 & & & & c_2 & c_1 \\ c_1 & c_0 & c_2 & c_1 & & & & c_2 \\ c_2 & c_1 & c_0 & c_2 & c_1 & & & \\ & c_2 & c_1 & c_0 & c_2 & c_1 & & \\ & & c_2 & c_1 & c_0 & c_2 & c_1 & \\ & & & c_2 & c_1 & c_0 & c_2 & c_1 \\ c_1 & & & & c_2 & c_1 & c_0 & c_2 \\ c_2 & c_1 & & & & c_2 & c_1 & c_0 \end{pmatrix}.$$

Hence for **every** $n \times n$ circulant matrix, whether $n = 2^m$ or not, we can multiply with a vector in $O(n \log_2 n)$ flops.

9.2 Toeplitz structure

A bit more common than periodic convolutions is ordinary convolutions and these are related to Toeplitz matrices.

A matrix T is said to be **Toeplitz** if

$$T_{ij} = t_{i-j}$$

for some column vector t indexed over the integers. We will denote this as

$$T = \text{Toep}[t] = \begin{pmatrix} t_0 & t_{-1} & t_{-2} & \cdots \\ t_1 & t_0 & t_{-1} & \ddots \\ t_2 & t_1 & \ddots & \ddots \\ \vdots & \ddots & \ddots & \ddots \end{pmatrix}.$$

A matrix H is said to be **Hankel** if

$$H_{ij} = h_{i+j}$$

for some column vector h indexed over the integers. We will denote this as

$$H = \text{Hank}[h] = \begin{pmatrix} h_0 & h_1 & h_2 & \cdots \\ h_1 & h_2 & \iddots & \\ h_2 & \iddots & \iddots & \\ \vdots & & & \end{pmatrix}.$$

Let $E[n]$ denote the $n \times n$ Hankel matrix with one's on the anti-diagonal and zeros every where else.

Note that

$$\text{Circ}[c] = \text{Toep}\left[\begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \\ c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix}\right].$$

Also note that multiplying by E converts a Toeplitz matrix to a Hankel matrix.

A structure to note is that

$$\text{Diag}\Big[\exp\Big(-\frac{\pi\iota}{n}i^2\Big)\Big]_{i=0}^{n-1} F[n] \,\text{Diag}\Big[\exp\Big(-\frac{\pi\iota}{n}j^2\Big)\Big]_{j=0}^{n-1} = \Big(\ \exp\big(-\tfrac{\pi\iota}{n}(i-j)^2\big)\ \Big)_{\{i,j=0\}}^{n-1}$$

which is Toeplitz. So the Fourier matrix can be row and column scaled to reveal a Toeplitz matrix.

9.2.1 Fast Toeplitz-vector multiply

To multiply a $m \times n$ Toeplitz matrix with a column vector use the circulant embedding:

$$\text{Circ}[t]\begin{pmatrix} x \\ 0 \end{pmatrix} = \begin{pmatrix} * \\ \text{Toep}[t]\, x \end{pmatrix}.$$

For example the 5×3 case:

$$\begin{pmatrix} t_{-2} & t_4 & t_3 & t_2 & t_1 & t_0 & t_{-1} \\ t_{-1} & t_{-2} & t_4 & t_3 & t_2 & t_1 & t_0 \\ t_0 & t_{-1} & t_{-2} & t_4 & t_3 & t_2 & t_1 \\ t_1 & t_0 & t_{-1} & t_{-2} & t_4 & t_3 & t_2 \\ t_2 & t_1 & t_0 & t_{-1} & t_{-2} & t_4 & t_3 \\ t_3 & t_2 & t_1 & t_0 & t_{-1} & t_{-2} & t_4 \\ t_4 & t_3 & t_2 & t_1 & t_0 & t_{-1} & t_{-2} \end{pmatrix}.$$

Therefore we can also multiply any $n \times n$ Toeplitz matrix into a column vector in $O(n \log_2 n)$ flops. Note that this embeds the Toeplitz matrix in the bottom left corner of a circulant matrix.

Also note that this means we can extend the FFT to any $n \neq 2^k$ because $F[n]$ can be written as diagonal times Toeplitz times diagonal in $O(n)$ flops. So we now have an FFT with no limitation on the size.

Everything extends to Hankel matrices because of $E[n]$.

9.3 Triangular Toeplitz matrices

Let

$$Z_{\downarrow}[n] = \begin{pmatrix} 0 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & \ddots & & \vdots \\ 0 & 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix}_{n \times n}$$

denote the shift-down matrix. Then a $n \times n$ lower triangular Toeplitz matrix can be written as

$$\text{Toep}[t] = \sum_{k=0}^{n-1} t_k \, Z_{\downarrow}^k[n] = \sum_{k=0}^{\infty} t_k \, Z_{\downarrow}^k[n],$$

which shows their close relationship with polynomials.

From this expression it is easy to see that the product of 2 lower triangular Toeplitz matrices is also lower triangular Toeplitz. In fact they can be computed rapidly as follows. If $a_k = b_k = c_k = 0$ for $0 > k$, and

$$\text{Toep}[a] \, \text{Toep}[b] = \text{Toep}[c],$$

then it follows that from

$$\begin{pmatrix} a_0 & 0 & \cdots & 0 \\ a_1 & a_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{n-1} & \cdots & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix},$$

c can be computed $O(n \log_2 n)$ flops using fast Toeplitz-vector multiplication.

Conversely if c is known then b can also be computed rapidly via fast forward substitution using the standard 2×2 recursive procedure with cost

$$f(n) \;\leqslant\; 2 f(n/2) + cn \log_2 n,$$

which yields a complexity of $O(n \log_2^2 n)$ flops. Note that the only reason the forward multiply is cheaper is because we first embed in a large circulant matrix, but that technique does not seem availble here, so one does need to be careful of the exact power of the logarithmic factor in these estimates.

9.4 Fast solvers

From equation 9.1.2 it is obvious that we can solve $\mathrm{Circ}[c]x = b$ in $O(n \log_2 n)$ flops, where $b \in \mathbb{R}^n$. However $\mathrm{Toep}[t]\ x = b$ seems harder to solve quickly by embedding techniques, except for the triangular case which is more straight forward.

Let $Z_\uparrow = Z_\downarrow^T$ denote the up-shift matrix. These two matrices don't exactly commute, but do up to low-rank. It is easy to see (exercise) that

$$\mathrm{rank}(Z_\uparrow^p Z_\downarrow^q - Z_\downarrow^q Z_\uparrow^p) \leqslant 2 \min(p, q).$$

The difficulty of fast algebraic algorithms for finite dimensional Toeplitz matrices can be related to this issue.

10 Fast Polynomial Arithmetic

Circulant and triangular Toeplitz matrices are closely related to polynomial arithmetic as they themselves are polynomials in the "Z" matrix. We only point out some simple connections here. Note that this is a vast field and we make no effort to be comprehensive or modern.

To the polynomial

$$p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$$

we associate the lower banded Toeplitz matrix

$$\mathrm{LToep}[p] = \begin{pmatrix} p_0 & 0 & \cdots \\ p_1 & p_0 & \ddots \\ \vdots & p_1 & \ddots \\ p_n & & \ddots \\ 0 & p_n & \\ \vdots & \ddots & \ddots \end{pmatrix}.$$

Note that we do not associate a size with this matrix. Then the product of 2 polynomials can be represented as

$$\begin{pmatrix} a_0 & 0 & \cdots \\ a_1 & a_0 & \ddots \\ \vdots & a_1 & \ddots \\ a_n & & \ddots \\ 0 & a_n & \\ \vdots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} b_0 & 0 & \cdots \\ b_1 & b_0 & \ddots \\ \vdots & b_1 & \ddots \\ b_m & & \ddots \\ 0 & b_m & \\ \vdots & \ddots & \ddots \end{pmatrix} = \begin{pmatrix} c_0 & 0 & \cdots \\ c_1 & c_0 & \ddots \\ \vdots & c_1 & \ddots \\ c_{n+m} & & \ddots \\ 0 & c_{n+m} & \\ \vdots & \ddots & \ddots \end{pmatrix}.$$

Note that c can be computed from a and b in $O(\max(n, m) \log_2(\max(m, n)))$ flops as the product of 2 lower triangular Toeplitz matrices.

The problem of polynomial division can be presented as the converse problem

$$\begin{pmatrix} a_0 & 0 & \cdots \\ a_1 & a_0 & \ddots \\ \vdots & a_1 & \ddots \\ a_{n-1} & \vdots & \ddots \\ a_n & a_{n-1} & \\ 0 & a_n & \ddots \\ \vdots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} b_0 & 0 & \cdots \\ b_1 & b_0 & \ddots \\ \vdots & b_1 & \ddots \\ b_m & \vdots & \ddots \\ 0 & b_m & \\ \vdots & \ddots & \ddots \end{pmatrix} + \begin{pmatrix} r_0 & 0 & \cdots \\ r_1 & r_0 & \ddots \\ \vdots & r_1 & \ddots \\ r_{n-1} & \vdots & \ddots \\ 0 & r_{n-1} & \\ \vdots & \ddots & \ddots \end{pmatrix} = \begin{pmatrix} c_0 & 0 & \cdots \\ c_1 & c_0 & \ddots \\ \vdots & c_1 & \ddots \\ c_{n-1} & \vdots & \ddots \\ c_n & c_{n-1} & \\ c_{n+1} & c_n & \ddots \\ \vdots & c_{n+1c} & \ddots \\ c_{n+m} & \vdots & \ddots \\ 0 & c_{n+m} & \\ \vdots & \ddots & \ddots \end{pmatrix},$$

where we are given the polynomial c and the divisor a, and the goal is to find the quotient b and the residual r. If we drop the first n rows and keep only $m+1$ rows after that we get

$$\begin{pmatrix} a_n & a_{n-1} & \\ 0 & a_n & \ddots \\ \vdots & \ddots & \ddots \end{pmatrix} \begin{pmatrix} b_0 & 0 & \cdots \\ b_1 & b_0 & \ddots \\ \vdots & b_1 & \ddots \\ b_m & \vdots & \ddots \\ 0 & b_m & \\ \vdots & \ddots & \ddots \end{pmatrix} + \begin{pmatrix} 0 & r_{n-1} & & \\ \vdots & \ddots & \ddots & \end{pmatrix} = \begin{pmatrix} c_n & c_{n-1} & \\ c_{n+1} & c_n & \ddots \\ \vdots & c_{n+1c} & \ddots \\ c_{n+m} & \vdots & \ddots \end{pmatrix},$$

and we see that the first column gives us the quotient b in $O(m \log_2^2 m)$ flops. Then the residual can be computed by fast polynomial multiplication of the first n rows of the above equation. Note that the existence of b and r follows from the known facts about lower triangular Toeplitz matrices (or from standard polynomial algebra).

The problem of fast polynomial evaluation at multiple points and the converse problem of finding the polynomial from its evaluation at multiple points we set aside for now and go back to the question of representation of Toeplitz matrices and their associated displacement structures.

11 Iterative methods

We say how easy it can be to design fast matrix-vector multiplication routines for several classes of matrices. Naturally the question arises whether that alone can be used to speed up the solution of $Ax = b$, to something faster than $O(n^3)$ flops. The answer is in principle *yes*, but it does not always yield the fastest possible solver. We show one such technique here.

Let $A = A^T$ be non-singular. Given the RHS b consider the associated Krylov matrix

$$K[A, b] = \begin{pmatrix} b & Ab & A^2 b & \cdots & A^{n-1} b \end{pmatrix}.$$

We will make the generic assumption that $K[A, b]$ is also non-singular (so b is not special). Then we look for a solution $x = K[A, b]\, z$ and observe that

$$K^T[A, b]\, A\, K[A, b] = \begin{pmatrix} b^T Ab & b^T A^2 b & b^T A^3 b & b^T A^4 b & \cdots \\ b^T A^2 b & b^T A^3 b & \ddots & \ddots & \\ b^T A^3 b & \ddots & \ddots & & \\ b^T A^4 b & \ddots & & & \\ \vdots & & & & \end{pmatrix},$$

is a Hankel matrix. So if we can solve the Hankel system

$$K^T[A,b]\,AK[A,b]z = K^T[A,b]\,b$$

fast, say in $O(n\log_2^2 n)$ flops, then the total cost of the algorithm will be at most the cost of $O(n)$ matrix-vector multiplies.

In a true iterative method the solution (or at least the residual) is tracked for smaller leading principal sub-matrices of the Hankel block and sometimes you can get early convergence, requiring much less than $O(n)$ multiplies. This savings unfortunately depends on the spectral properties of A making it hard to control in practice.

However, in practice, the primary reason for using such iterative methods is to avoid the large amounts of fill-in during Gaussian elimination of sparse matrices.

12 Displacement structure

We mentioned earlier that Toeplitz matrices can be written as polynomials in $Z_\downarrow$ and its tranpose $Z_\uparrow$. However, in the finite dimensional case, these two matrices do not commute. So this leads us to examine the commutator of a Toeplitz matrix $T = \mathrm{Toep}[t]$ with $Z_\downarrow$:

$$Z_\downarrow T - TZ_\downarrow = \begin{pmatrix} -t_{-1} & -t_{-2} & \cdots & -t_{-(n-1)} & 0 \\ 0 & 0 & 0 & \cdots & t_{-(n-1)} \\ 0 & 0 & \ddots & \ddots & t_{-(n-2)} \\ \vdots & \vdots & \ddots & & \vdots \\ 0 & 0 & & 0 & t_{-1} \end{pmatrix},$$

which is rank 2. However note that the RHS has lost t_i for $0 \leqslant i$ as the operator $L[T] = Z_\downarrow T - TZ_\downarrow$ has the lower triangular Toeplitz matrices in its nullspace. Instead we look at the closely allied operator

$$T - Z_\downarrow TZ_\uparrow = \begin{pmatrix} t_0 & t_{-1} & t_{-2} & \cdots \\ t_1 & 0 & 0 & \cdots \\ t_2 & 0 & \ddots & \\ \vdots & \vdots & & \end{pmatrix},$$

which is also rank 2 but has the virtue of fully specifying T just from the RHS. In particular the Sylvester-type operator $L[T] = T - Z_\downarrow TZ_\uparrow$ is invertible.

However what is truly amazing is how well these operators interact with Gaussian elimination. It helps to be a bit more general first. So we consider *displacement* operators of the form

$$L[F,G][A] = A - FAG^T,$$

where to ensure easy invertibility of $L[F,G]$ we assume that F and G are lower-triangular matrices and that $F_{i,i}G_{j,j} \neq 1$ for all i,j. It is easy to check that this guarantees that $L[F,G]$ is non-singular. We are interested in the case when $L[F,G][A] = XY^T$ is a low-rank (compared to the size of A) matrix. In particular we want to know if the Schur complement of A has a similar structure.

12.1 Sylvester type equations

We first step aside to look at linear operators of the form

$$L[A,B][X] = X - AXB^T = Y.$$

If $X \in \mathbb{R}^{n\times n}$ then converting this into a standard form of the type "$Ax = b$" and then solving via Gaussian elimination would lead to $O(n^6)$ flops. By using the "iterative" method trick we can reduce it to $O(n^5)$ flops. If A and B are structured matrices such that AXB^T can be computed in $O(n^\alpha)$ flops, then the cost would actually be $O(n^{\alpha+2})$ flops. However, if we can compute the spectral decompositions of A and B then any further operations can be reduced to $O(n^3)$ flops.

To see this we assume that both A and B are lower triangular matrices. Then:

$$\begin{aligned}
\begin{pmatrix} \xi & x_2^T \\ x_1 & X_2 \end{pmatrix} - \begin{pmatrix} \alpha & 0 \\ b & A_2 \end{pmatrix}\begin{pmatrix} \xi & x_2^T \\ x_1 & X_2 \end{pmatrix}\begin{pmatrix} \beta & c^T \\ 0 & B_2^T \end{pmatrix} &= \begin{pmatrix} \psi & y_2^T \\ y_1 & Y_2 \end{pmatrix} \\
\begin{pmatrix} \xi & x_2^T \\ x_1 & X_2 \end{pmatrix} - \begin{pmatrix} \alpha\xi & \alpha x_2^T \\ b\xi + A_2 x_1 & bx_2^T + A_2 X_2 \end{pmatrix}\begin{pmatrix} \beta & c^T \\ 0 & B_2^T \end{pmatrix} &= \begin{pmatrix} \psi & y_2^T \\ y_1 & Y_2 \end{pmatrix} \\
\begin{pmatrix} \xi & x_2^T \\ x_1 & X_2 \end{pmatrix} - \begin{pmatrix} \alpha\xi\beta & \alpha(\xi c^T + x_2^T B_2^T) \\ (b\xi + A_2 x_1)\beta & (b\xi + A_2 x_1)c^T + (bx_2^T + A_2 X_2)B_2^T \end{pmatrix} &= \begin{pmatrix} \psi & y_2^T \\ y_1 & Y_2 \end{pmatrix},
\end{aligned}$$

and we can construct a recursive algorithm as follows:

$$\begin{aligned}
\xi &= \frac{\psi}{1-\alpha\beta} \\
x_1 &= (I - \beta A_2)^{-1}(y_1 + b\xi\beta) \\
x_2 &= (I - \alpha B_2)^{-1}(y_2 + c\xi\alpha) \\
X_2 - A_2 X_2 B_2^T &= Y_2 + \xi b c^T + A_2 x_1 c^T + b x_2^T B_2^T.
\end{aligned}$$

One can see that as long as $\alpha\beta \neq 1$ there exists a unique solution and we can now find it in $O(n^3)$ flops. However this trick of reducing the problem to the case when A and B are triangular breaks down when there are more terms:

$$X - AXB^T - CXD^T = Y.$$

In this case we need to assume that A, B, C, D are already lower-triangular. In fact understanding the structure of this linear operator remains a significant open problem.

12.2 Schur algorithm

Now we turn our attention back to displacement structured matrices (equation 12) and their fast LU factorization.

Block-partitoning the equation we see that

$$\begin{aligned}
\begin{pmatrix} \alpha & b^T \\ c & D \end{pmatrix} - \begin{pmatrix} \varphi & 0 \\ f & F_2 \end{pmatrix}\begin{pmatrix} \alpha & b^T \\ c & D \end{pmatrix}\begin{pmatrix} \gamma & g^T \\ 0 & G_2^T \end{pmatrix} &= \begin{pmatrix} x_1^T \\ X_2 \end{pmatrix}\begin{pmatrix} y_1 & Y_2^T \end{pmatrix} \\
\begin{pmatrix} \alpha & b^T \\ c & D \end{pmatrix} - \begin{pmatrix} \varphi\alpha & \varphi b^T \\ f\alpha + F_2 c & fb^T + F_2 D \end{pmatrix}\begin{pmatrix} \gamma & g^T \\ 0 & G_2^T \end{pmatrix} &= \begin{pmatrix} x_1^T \\ X_2 \end{pmatrix}\begin{pmatrix} y_1 & Y_2^T \end{pmatrix} \\
\begin{pmatrix} \alpha & b^T \\ c & D \end{pmatrix} - \begin{pmatrix} \varphi\alpha\gamma & \varphi\alpha g^T + \varphi b^T G_2^T \\ f\alpha\gamma + F_2 c\gamma & f\alpha g^T + F_2^T c g^T + f b^T G_2^T + F_2 D G_2^T \end{pmatrix} &= \begin{pmatrix} x_1^T \\ X_2 \end{pmatrix}\begin{pmatrix} y_1 & Y_2^T \end{pmatrix} \\
\begin{pmatrix} \alpha - \varphi\alpha\gamma & b^T - \varphi\alpha g^T - \varphi b^T G_2^T \\ c - f\alpha\gamma - F_2 c\gamma & D - f\alpha g^T - F_2^T c g^T - f b^T G_2^T - F_2 D G_2^T \end{pmatrix} &= \begin{pmatrix} x_1^T y_1 & x_1^T Y_2^T \\ X_2 y_1 & X_2 Y_2^T \end{pmatrix}.
\end{aligned}$$

Therefore, we can recover the first row and column of A from F, G, X, Y, as usual, via

$$\begin{aligned}
\alpha &= \frac{x_1^T y_1}{1-\varphi\gamma} \\
c &= (I - F_2\,\gamma)^{-1}(X_2\,y_1 + f\alpha\gamma) \\
b &= (I - G_2\,\varphi)^{-1}(Y_2\,x_1 + g\alpha\varphi).
\end{aligned}$$

The cost of this recovery depends on the structure of the two lower triangular matrices F_2 and G_2. We will assume that these are no more than $O(n)$ flops. Note that this is true for the Toeplitz case as both F_2, G_2 are then bi-diagonal. But many other structured matrices can be imagined for F and G.

Now we do 1 step of Gaussian elimination assuming $\alpha \neq 0$. Applying the elementary Gauss transform to the left of the displacement equation for A we obtain:

$$\begin{pmatrix} 1 & 0 \\ \frac{-c}{\alpha} & I \end{pmatrix}\begin{pmatrix} \alpha & b^T \\ c & D \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ \frac{-c}{\alpha} & I \end{pmatrix}\begin{pmatrix} \varphi & 0 \\ f & F_2 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ \frac{c}{\alpha} & I \end{pmatrix}\begin{pmatrix} 1 & 0 \\ \frac{-c}{\alpha} & I \end{pmatrix}\begin{pmatrix} \alpha & b^T \\ c & D \end{pmatrix}\begin{pmatrix} \gamma & g^T \\ 0 & G_2^T \end{pmatrix} =$$

$$\begin{pmatrix} \alpha & b^T \\ 0 & D - \frac{cb^T}{\alpha} \end{pmatrix} - \begin{pmatrix} \varphi & 0 \\ f - \frac{c\varphi}{\alpha} & F_2 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ \frac{c}{\alpha} & I \end{pmatrix}\begin{pmatrix} \alpha & b^T \\ 0 & D - \frac{cb^T}{\alpha} \end{pmatrix}\begin{pmatrix} \gamma & g^T \\ 0 & G_2^T \end{pmatrix} =$$

$$\begin{pmatrix} \alpha & b^T \\ 0 & D - \frac{cb^T}{\alpha} \end{pmatrix} - \begin{pmatrix} \varphi & 0 \\ f - \frac{c\varphi}{\alpha} + \frac{F_2 c}{\alpha} & F_2 \end{pmatrix}\begin{pmatrix} \alpha & b^T \\ 0 & D - \frac{cb^T}{\alpha} \end{pmatrix}\begin{pmatrix} \gamma & g^T \\ 0 & G_2^T \end{pmatrix} =$$

$$\begin{pmatrix} \alpha & b^T \\ 0 & S \end{pmatrix} - \begin{pmatrix} \varphi & 0 \\ f - \frac{c\varphi}{\alpha} + \frac{F_2 c}{\alpha} & F_2 \end{pmatrix}\begin{pmatrix} \alpha & b^T \\ 0 & S \end{pmatrix}\begin{pmatrix} \gamma & g^T \\ 0 & G_2^T \end{pmatrix} =$$

$$\begin{pmatrix} \alpha & b^T \\ 0 & S \end{pmatrix} - \begin{pmatrix} \varphi & 0 \\ f - \frac{c\varphi}{\alpha} + \frac{F_2 c}{\alpha} & F_2 \end{pmatrix}\begin{pmatrix} \alpha\gamma & \alpha g^T + b^T G_2^T \\ 0 & SG_2 \end{pmatrix} =$$

$$\begin{pmatrix} \alpha & b^T \\ 0 & S \end{pmatrix} - \begin{pmatrix} \varphi\alpha\gamma & \varphi\alpha g^T + \varphi b^T G_2^T \\ f\alpha\gamma - c\varphi\gamma + F_2\,c\gamma & \left(f - \frac{c\varphi}{\alpha} + \frac{F_2 c}{\alpha}\right)(\alpha g^T + b^T G_2^T) + F_2\,SG_2 \end{pmatrix},$$

and the RHS becomes

$$\begin{aligned}
\begin{pmatrix} 1 & 0 \\ \frac{-c}{\alpha} & I \end{pmatrix}\begin{pmatrix} x_1^T \\ X_2 \end{pmatrix}\begin{pmatrix} y_1 & Y_2^T \end{pmatrix} &= \begin{pmatrix} x_1^T \\ X_2 - \frac{cx_1^T}{\alpha} \end{pmatrix}\begin{pmatrix} y_1 & Y_2^T \end{pmatrix} \\
&= \begin{pmatrix} x_1^T y_1 & x_1^T Y_2^T \\ \left(X_2 - \frac{cx_1^T}{\alpha}\right)y_1 & \left(X_2 - \frac{cx_1^T}{\alpha}\right)Y_2^T \end{pmatrix}.
\end{aligned}$$

Note that the $(2,1)$ entry implies that

$$-(f\alpha\gamma - c\varphi\gamma + F_2\,c\gamma) = \left(X_2 - \frac{cx_1^T}{\alpha}\right)y_1.$$

So a displacement equation for the Schur complement $S = D - \frac{cb^T}{\alpha}$ can be obtained as

$$\begin{aligned}
S - F_2 S G_2^T &= \left(X_2 - \frac{cx_1^T}{\alpha}\right)Y_2^T + \left(f - \frac{c\varphi}{\alpha} + \frac{F_2 c}{\alpha}\right)(\alpha g^T + b^T G_2^T) \\
&= \left(X_2 - \frac{cx_1^T}{\alpha}\right)Y_2^T - \frac{1}{\alpha\gamma}\left(X_2 - \frac{cx_1^T}{\alpha}\right)y_1(\alpha g^T + b^T G_2^T) \\
&= \left(X_2 - \frac{cx_1^T}{\alpha}\right)\left(Y_2^T - \frac{1}{\alpha\gamma}y_1(\alpha g^T + b^T G_2^T)\right)
\end{aligned}$$

which has the same rank as the displacement of A, provided $\gamma \neq 0$.

If $\gamma = 0$ then from equation 12.2 we also have that

$$\left(X_2 - \frac{cx_1^T}{\alpha}\right) y_1 = 0,$$

which implies that

$$X_2 - \frac{cx_1^T}{\alpha},$$

is rank deficient and hence the rank of

$$\left(X_2 - \frac{cx_1^T}{\alpha}\right) Y_2^T + \left(f - \frac{c\varphi}{\alpha} + \frac{F_2 c}{\alpha}\right)(\alpha g^T + b^T G_2^T),$$

is no more than the displacement rank of A!

So in every case the displacement rank of S is no more than the displacement rank of A. Since each step of this implicit Gaussian elimination on A takes $O(n)$ flops, the entire LU factorization of A can be computed in $O(n^2)$ algorithms. This is referred to as Schur's algorithm and is an effective way of reducing the cost of Gaussian elimination from $O(n^3)$ flops to $O(n^2)$ flops for matrices with an efficient displacement equation. By this we mean that F, G must be lower triangular and cheap to multiply and do forward substitution on (say $O(n)$ flops), and the correspoding displacement rank must be small.

Note that this is indeed true for Toeplitz and Hankel matrices, but there are significantly more matrices for which this is true.

The Schur algorithm as stated is not numerically stable. However, due to lack of time, we ignore those issues here, and just point out that they will be eventually resolved by another algorithm in many cases.

12.3 Examples

We now present other (than Toeplitz and Hankel) examples of small displacement rank. Consider:

$$\begin{pmatrix} X_1^0 & X_1^1 & X_1^2 & \cdots \\ X_2^0 & X_2^1 & X_2^2 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix} - \mathrm{Diag}[(X_i)_{i=1}] \begin{pmatrix} X_1^0 & X_1^1 & X_1^2 & \cdots \\ X_2^0 & X_2^1 & X_2^2 & \cdots \\ \vdots & \vdots & \vdots & \end{pmatrix} Z_\uparrow = \begin{pmatrix} X_1^0 & 0 & \cdots \\ X_2^0 & 0 & \cdots \\ \vdots & \vdots & \end{pmatrix},$$

which has rank p when $X_i \in \mathbb{R}^{p \times p}$ and $Z_\uparrow$ is block shift-up matrix. If X_i is 1×1 then we get the standard Vandermonde matrix.

Consider a family of matrices $A_{i,j}$ that satisfy a family of Sylvester-type equations:

$$A_{i,j} - F_i A_{i,j} G_j = X_i Y_j^T,$$

where we assume that the above linear operators are all non-singular. Then

$$A - \mathrm{Diag}[(F_i)_i] A \,\mathrm{Diag}[(G_j)_j] = XY^T$$

has small displacement rank. If $A_{i,j} \in \mathbb{R}^{1 \times 1}$, then the family is referred to as generalized Pick matrices:

$$A_{i,j} = \frac{X_{i,:} Y_{j,:}^T}{1 - F_i G_j},$$

and are closely related to generalized Cauchy matrices which arise from looking at the related displacement operator

$$FA - AG = XY^T.$$

From a theoretical perspective there is not much difference between these 2 formulations and one can look at the even more general formulation

$$L[Z] = AZB^T + CZD^T = XY^T.$$

However this just makes the formulas unnecessarily complicated at this stage.

Generalized Pick (Cauchy) matrices allow (row and column) **pivoting** during the Schur algorithm, which allows these algorithms to be effectively numerically stable. This is an important consideration in practice.

The generalized Pick (Cauchy) matrices have one other important property. Certain of their submatrices have low numerical rank, which is not exploited by the Schur algorithm. This observation will eventually lead to fast $O(n \log_2^2 n)$ approximate solvers for Toeplitz type matrices that are also numerically stable.

There are also other ways to get low displacement ranks in practice. Note that

$$\begin{pmatrix} A^{-1} & G^T \\ F & A \end{pmatrix} = \begin{pmatrix} I & 0 \\ FA & I \end{pmatrix} \begin{pmatrix} A^{-1} & G^T \\ 0 & A - FAG^T \end{pmatrix}$$
$$= \begin{pmatrix} I & G^T A^{-1} \\ 0 & I \end{pmatrix} \begin{pmatrix} A^{-1} - G^T A^{-1} F & 0 \\ F & A \end{pmatrix}.$$

The last matrix on both equations must have the same rank, which proves that A^{-1} has the same displacement rank as A.

If $G^T F - I$ is low rank then one can show that the product of 2 low displacement rank matrices for this displacement operator will also have low rank (but larger than the individual displacement ranks). So for example the product of 2 Toeplitz matrices has low displacement rank even though they are not Toeplitz.

In general there are lots of matrices with good low displacement rank structure which are hard to identify directly. For example if both F and G are lower triangular Toeplitz matrices then clearly we have an associated fast Schur algorithm. However there seems to be no simple characterization of the explicit structure of the matrix itself.

There is an important class of examples associated to Toeplitz structures which cannot be handled currently by displacement rank approach. Define the Kronecker product of two matrices:

$$A \otimes B = \begin{pmatrix} A_{1,1} B & A_{1,2} B & \dots \\ A_{2,1} B & A_{2,2} B & \dots \\ \vdots & \vdots & \end{pmatrix}.$$

Then if T_1 and T_2 are 2 Toeplitz matrices, $T_1 \otimes T_2$ is an example of a doubly Toeplitz matrix; that is a block Toeplitz matrix where each block is also Toeplitz. In the above case the simple tensor structure gives access to a simple fast direct solver. However for more general doubly Toeplitz matrices no good displacement structure is known. Similarly one can look at triply Toeplitz matrices and even less is known about them. These higher order Toeplitz matrices arise in non 1D signal processing problems and form a huge open problem. Some success has been obtained on this class of problems by first converting Toeplitz structure to Pick (Cauchy) structure and exploiting the numerical low-rank in their sub-matrices. Toeplitz matrices can be converted to Pick (Cauchy) like matrices by means of the FFT (and other fast trigonometric transforms).

13 Low-rank structures

Observe that if $T = \text{Toep}[t]$ then

$$T - ZTZ^H = XY^H,$$

is of rank 4. We already know that

$$Z[n] = F[n]\,\Lambda[n]\,F^H[n].$$

Therefore

$$F^H TF - \Lambda\,(F^H TF)\,\Lambda^H = (F^H X)\,(F^H Y)^H = AB^H$$

Note that this displacement operator is **not** invertible, but the off-diagonal entries can be computed readily from the RHS:

$$(F^H TF)_{i,j} = \frac{A_{i,:}\,B^H_{j,:}}{1 - \Lambda_i\,\hat{\Lambda}_j},$$

as

$$\Lambda_i\,\overline{\Lambda}_j = \omega_n^{i-j} \neq 1$$

if $0 \leqslant i \neq j < n$. The diagonal entries are well-defined but have to be computed separately. Note that

$$\begin{aligned}
(F^H[n]\,TF[n])_{i,i} &= \sum_{p,q=0}^{n-1} \omega_n^{-ip} T_{p,q}\,\omega_n^{qi} \\
&= \sum_{p,q=0}^{n-1} \omega_n^{-i(p-q)} t_{p-q} \\
&= n\omega_n^{0i} t_0 + (n-1)\omega_n^{-i} t_1 + (n-2)\omega_n^{-2i} t_2 + \cdots + \\
&\quad + (n-1)\omega_n^{i} t_{-1} + (n-2)\omega_n^{2i} t_{-2} + \cdots +
\end{aligned}$$

We see that all of these numbers can be computed at the cost of 2 FFT's plus $O(n)$ flops.

In summary we can rapidly convert a Toeplitz matrix to a Pick-type matrix.

Other examples that we want to keep in mind arise from potential theory problems. Let $x_i \in \mathbb{R}^3$ denote the position of n points of mass m_i. Then the gravitational potential at $y \in \mathbb{R}^3$ is given by

$$p(y) = \sum_{i=1}^{n} \frac{m_i}{\|x_i - y\|}.$$

Given n points $y_i \in \mathbb{R}^3$ can we rapidly compute $p(y_i)$? Given $p(y_i)$ can we rapidly find m_i? Note that the above equations can be represented as

$$\begin{pmatrix} \frac{1}{\|y_1 - x_1\|} & \frac{1}{\|y_1 - x_2\|} & \cdots \\ \frac{1}{\|y_2 - x_1\|} & \frac{1}{\|y_2 - x_2\|} & \cdots \\ \vdots & \vdots & \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} p(y_1) \\ p(y_2) \\ \vdots \end{pmatrix}.$$

What is a simple, but useful, approximate structure in these types of matrices is that large submatrices can have small numerical rank. For example in the above case it is easy to check that if

$$\min_{i,j} \|x_i - y_j\| \gg \max_{i,j} (\|x_i\| + \|y_j\|),$$

then the matrix will have small numerical rank and a low-rank expansion can be computed using classical multi-pole expansions.

13.1 Fast multiplication leads to fast inversion

Low-rank matrices are pleasantly simple to work with. Consider for example computing Ax where $A = \mathrm{Diag}[d] + u\omega v^T$.

Note that we can proceed as follows:

$$\begin{aligned} g &= v^T x \\ h &= \omega g \\ b_i = (Ax)_i &= d_i x_i + u_i h \end{aligned}$$

and compute $b = Ax$ in $O(n)$ flops. We can also rapidly compute x given b by using the above equations. First we re-write the above equations in **sparse** matrix form:

$$\begin{pmatrix} 1 & 0 & -v^T & 0 \\ -\omega & 1 & 0 & 0 \\ 0 & -u & -\mathrm{Diag}[d] & I \end{pmatrix} \begin{pmatrix} g \\ h \\ x \\ b \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

So fast multiplication requires us to *solve* the above equations for g, h, b:

$$\begin{pmatrix} 1 & 0 & 0 \\ -\omega & 1 & 0 \\ 0 & -u & I \end{pmatrix} \begin{pmatrix} g \\ h \\ b \end{pmatrix} = \begin{pmatrix} v^T x \\ 0 \\ \mathrm{Diag}[d]\, x \end{pmatrix}.$$

We see that the above equations are sparse lower triangular and we can do forward substitution in $O(n)$ flops and recover the same algorithm as before. However if we want to solve $Ax = b$ for x instead, then we can re-arrange the sparse system to solve for g, h, x instead:

$$\begin{pmatrix} 1 & 0 & -v^T \\ -\omega & 1 & 0 \\ 0 & u & \mathrm{Diag}[d] \end{pmatrix} \begin{pmatrix} g \\ h \\ x \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ b \end{pmatrix}.$$

Note the system is not triangular any more, and if we do block Gaussian elimination we will return to the original system of equations:

$$\begin{aligned} \begin{pmatrix} 1 & 0 & -v^T \\ -\omega & 1 & 0 \\ 0 & u & \mathrm{Diag}[d] \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 \\ -\omega & 1 & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} 1 & 0 & -v^T \\ 0 & 1 & -\omega v^T \\ 0 & u & \mathrm{Diag}[d] \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ -\omega & 1 & 0 \\ 0 & u & I \end{pmatrix} \begin{pmatrix} 1 & 0 & -v^T \\ 0 & 1 & -\omega v^T \\ 0 & 0 & \mathrm{Diag}[d] + u\omega v^T \end{pmatrix}. \end{aligned}$$

So we gain nothing from this approach. However remember that sparse Gaussian elimination is sensitive to the ordering of the rows and columns, so let's try a different order. Note that in the current order the variables will be produced in the sequence x, h, g. Let's try the order h, g, x instead, and also exchange the first and third equations:

$$\begin{pmatrix} \mathrm{Diag}[d] & u & 0 \\ 0 & 1 & -\omega \\ -v^T & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ h \\ g \end{pmatrix} = \begin{pmatrix} b \\ 0 \\ 0 \end{pmatrix}.$$

We now repeat our block Gaussian elimination:

$$\begin{pmatrix} \mathrm{Diag}[d] & u & 0 \\ 0 & 1 & -\omega \\ -v^T & 0 & 1 \end{pmatrix} =$$

$$\begin{pmatrix} I & 0 & 0 \\ 0 & 1 & 0 \\ -v^T\mathrm{Diag}^{-1}[d] & 0 & 1 \end{pmatrix}\begin{pmatrix} \mathrm{Diag}[d] & u & 0 \\ 0 & 1 & -\omega \\ 0 & v^T\,\mathrm{Diag}^{-1}[d]\,u & 1 \end{pmatrix} =$$

$$\begin{pmatrix} I & 0 & 0 \\ 0 & 1 & 0 \\ -v^T\mathrm{Diag}^{-1}[d] & v^T\,\mathrm{Diag}^{-1}[d]\,u & 1 \end{pmatrix}\begin{pmatrix} \mathrm{Diag}[d] & u & 0 \\ 0 & 1 & -\omega \\ 0 & 0 & 1+\omega v^T\,\mathrm{Diag}^{-1}[d]\,u \end{pmatrix}.$$

Note that this time we compute the LU factorization in $O(n)$ flops as we only have $O(1)$ fill-in. Furthermore we can now compute h, g, x in $O(n)$ flops too.

If we push through and do the forward substitution we get

$$\begin{pmatrix} I & 0 & 0 \\ 0 & 1 & 0 \\ -v^T\,\mathrm{Diag}^{-1}[d] & v^T\,\mathrm{Diag}^{-1}[d]\,u & 1 \end{pmatrix}\begin{pmatrix} b \\ 0 \\ v^T\,\mathrm{Diag}^{-1}[d]\,b \end{pmatrix} = \begin{pmatrix} b \\ 0 \\ 0 \end{pmatrix}$$

and then follow up with back substitution to get

$$\begin{pmatrix} \mathrm{Diag}[d] & u & 0 \\ 0 & 1 & -\omega \\ 0 & 0 & 1+\omega v^T\,\mathrm{Diag}^{-1}[d]\,u \end{pmatrix}\begin{pmatrix} \mathrm{Diag}^{-1}[d]\Big(b - \frac{u\omega v^T\,\mathrm{Diag}^{-1}[d]\,b}{1+\omega v^T\,\mathrm{Diag}^{-1}[d]\,u}\Big) \\ \frac{\omega v^T\,\mathrm{Diag}^{-1}[d]\,b}{1+\omega v^T\,\mathrm{Diag}^{-1}[d]\,u} \\ \frac{v^T\,\mathrm{Diag}^{-1}[d]\,b}{1+\omega v^T\,\mathrm{Diag}^{-1}[d]\,u} \end{pmatrix} = \begin{pmatrix} b \\ 0 \\ v^T\,\mathrm{Diag}^{-1}[d]\,b \end{pmatrix}.$$

From this we get famous Sherman-Morrison-Woodbury formula

$$(\mathrm{Diag}[d] + u\omega v^T)^{-1} = \mathrm{Diag}^{-1}[d] - \frac{\mathrm{Diag}^{-1}[d]\,u\omega v^T\,\mathrm{Diag}^{-1}[d]}{1+\omega v^T\,\mathrm{Diag}^{-1}[d]\,u}.$$

Inspite of these formulas notes that the fast solver only required us to form the "lifted" sparse matrix in the right order and then just depend on sparse Gaussian elimination (for which there exist many highly developed codes).

This the main way in which fast direct solvers are currently implemented today and the core ideas can be traced to two sources. One is the body of work by Dewilde and van der Veen, and the other is the thesis of my student Tim Pals. We will follow this pattern in this class.

13.2 FMM

Above we looked at a fairly simple low-rank structure and observed that it was fairly trivial to derive fast algorithms provided one looked at the problem the right way. This approach can be extended now to more complex low-rank structures. The full complexity is captured in what I have habitually referred to as the Fast Multipole Method (FMM) structure, as this was technically well-known to Rokhlin and his group of collaborators early on, even though they did not seem to emphasize the algebraic aspects, being more concerned initally about analytic issues. The algebraic structures were most clearly realized and exploited in the afore mentioned work of Dewilde and van der Veen in the context of time varying systems theory. However it is unknown how much the latter work can be generalized to the full class of FMM matrices and remains one of the most important open problems in the field.

14 SSS

The FMM structure is concerned with off-diagonal blocks that have low-rank, so we first look at some of their simpler properties.

14.1 Basic observations

First note that such low-rank is preserved under addition:

$$\begin{pmatrix} * & UV^T \\ * & * \end{pmatrix} + \begin{pmatrix} * & PQ^T \\ * & * \end{pmatrix} = \begin{pmatrix} * & \begin{pmatrix} U & P \end{pmatrix} \begin{pmatrix} V^T \\ Q^T \end{pmatrix} \\ * & * \end{pmatrix}.$$

Next note that it is also preserved under multiplication:

$$\begin{pmatrix} A & UV^T \\ * & * \end{pmatrix} \begin{pmatrix} * & PQ^T \\ * & B \end{pmatrix} = \begin{pmatrix} * & \begin{pmatrix} U & AP \end{pmatrix} \begin{pmatrix} V^T B \\ Q^T \end{pmatrix} \\ * & * \end{pmatrix}.$$

It is also preserved under LU factorization:

$$\begin{pmatrix} A_1 & UV^T \\ PQ^T & A_2 \end{pmatrix} = \begin{pmatrix} I & 0 \\ PQ^T A_1^{-1} & I \end{pmatrix} \begin{pmatrix} A_1 & UV^T \\ 0 & A_2 - PQ^T A_1^{-1} UV^T \end{pmatrix}.$$

And finally it is preserved under inversion:

$$\begin{aligned} \begin{pmatrix} A_1 & UV^T \\ PQ^T & A_2 \end{pmatrix}^{-1} &= \begin{pmatrix} A_1^{-1} & -A_1^{-1} UV^T S^{-1} \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -PQ^T A_1^{-1} & I \end{pmatrix} \\ &= \begin{pmatrix} * & -A_1^{-1} UV^T S^{-1} \\ -S^{-1} PQ^T A_1^{-1} & * \end{pmatrix}. \end{aligned}$$

In fact neither LU factorization or inversion cause an increase in off-diaogonal ranks (called Schur rank from now on).

The next key observation is that for some matrices the ranks of all off-diagonal blocks can be small. So if you look at the block partitioning

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix},$$

then both upper off-diagonal blocks

$$\begin{pmatrix} A_{12} & A_{13} \end{pmatrix}, \qquad \begin{pmatrix} A_{13} \\ A_{23} \end{pmatrix},$$

might have low rank. How do we exploit this type of property in constructing fast algorithms?

First some examples. If we look at a banded matrix of bandwidth k,

$$A_{i,j} = 0, \qquad |i - j| > k,$$

then it is easy to check that all off-diagonal blocks have rank at most k. Now it follows from what we just proved that the inverse of such a matrix, though not banded, also has off-diagonal blocks of rank at most k. Furthermore if A and B are two banded matrices of bandwidth k, then $(A^{-1}+B^{-1})^{-1}$ also has Schur rank at most $2k$.

14.2 The representation

To exploit this property well we must have a matrix representation that is compact and amenable to fast matrix algorithms. The work Eidelman, Gohberg, Dewilde and van der Veen, made such representations available. We will refer to this as Sequentially Semi-Separable (SSS) representations.

An SSS representation of a matrix A is a set of sequences of matrices (D, U, V, W, P, Q, R) such that

$$A = \begin{pmatrix} D_1 & U_1 V_2^T & U_1 W_2 V_3^T & U_1 W_2 W_3 V_4^T & \cdots \\ P_2 Q_1^T & D_2 & U_2 V_3^T & U_2 W_3 V_4^T & \cdots \\ P_3 R_2 Q_1^T & P_3 Q_2^T & D_3 & U_3 V_4^T & \cdots \\ P_4 R_3 R_2 Q_1^T & P_4 R_3 Q_2^T & P_4 Q_3^T & D_4 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

Note that the general formula is:

$$\begin{aligned} A_{i,i} &= D_i \\ A_{i,j} &= U_i W_{i+1} \cdots W_{j-1} V_j^T, & i<j, \\ A_{i,j} &= P_i R_{i-1} \cdots R_{j+1} Q_j^T, & i>j. \end{aligned}$$

It is important to keep in mind that the matrices involved may not be square matrices. They point of tihs representation is that it immediately provides a low rank factorization of each off-diagonal block that touches the diagonal:

$$\begin{aligned} \begin{pmatrix} U_1 V_2^T & U_1 W_2 V_3^T & U_1 W_2 W_3 V_4^T & \cdots \end{pmatrix} &= U_1 \begin{pmatrix} V_2^T & W_2 V_3^T & W_2 W_3 V_4^T & \cdots \end{pmatrix} \\ \begin{pmatrix} U_1 W_2 V_3^T & U_1 W_2 W_3 V_4^T & \cdots \\ U_2 V_3^T & U_2 W_3 V_4^T & \cdots \end{pmatrix} &= \begin{pmatrix} U_1 W_1 \\ U_2 \end{pmatrix} \begin{pmatrix} V_3^T & W_3 V_4^T & \cdots \end{pmatrix} \\ \begin{pmatrix} U_1 W_2 W_3 V_4^T & \cdots \\ U_2 W_3 V_4^T & \cdots \\ U_3 V_4^T & \cdots \end{pmatrix} &= \begin{pmatrix} U_1 W_1 W_2 \\ U_2 W_1 \\ U_3 \end{pmatrix} \begin{pmatrix} V_4^T & \cdots \end{pmatrix}. \end{aligned}$$

We will refer to the off-diagonal blocks that appear on the LHS as Hankel blocks to denote they are special off-diagonal blocks that touch both the diagonal and corner of the parent matrix.

14.3 Construction

While it is clear that these representations provide a rank factorization of each Hankel block, it is not clear if they can all be the minimal low rank factorization simultaneously. Dewilde and van der Veen showed that this is indeed the case and provided an algorithm to construct the representation in $O(n^2)$ flops assuming the underlying matrix A is not special. If the matrix A is special, for example a generalized Pick (Cauchy) matrix, then FMM techniques (Greengard & Rokhlin, etc.) can be used to construct the SSS representation in $O(n \log_2^2 n)$ flops. Here we just describe the more general algorithm first.

First compute the low-rank factorization (say using an SVD):

$$\begin{pmatrix} A_{1,2} & A_{1,3} & \cdots \end{pmatrix} = U_1 H_1^T = U_1 \begin{pmatrix} H_{1,2}^T & H_{1,3}^T & \cdots \end{pmatrix}.$$

Then consider the next Hankel block:

$$\begin{aligned} \begin{pmatrix} A_{1,3} & A_{1,4} & \cdots \\ A_{2,3} & A_{2,4} & \cdots \end{pmatrix} &= \begin{pmatrix} U_1 & \\ & I \end{pmatrix} \begin{pmatrix} H_{1,3}^T & H_{1,4}^T & \cdots \\ A_{2,3} & A_{2,4} & \cdots \end{pmatrix} \\ &= \begin{pmatrix} U_1 & \\ & I \end{pmatrix} \begin{pmatrix} W_2 \\ U_2 \end{pmatrix} H_2^T \\ &= \begin{pmatrix} U_1 W_2 \\ U_2 \end{pmatrix} H_2^T \\ &= \begin{pmatrix} U_1 W_2 \\ U_2 \end{pmatrix} \begin{pmatrix} H_{2,3}^T & H_{2,4}^T & \cdots \end{pmatrix}. \end{aligned}$$

If we ensure that we low rank factorizations with the first factor having orthonormal columns:

$$U_1^T U_1 = I, \qquad \begin{pmatrix} W_2 \\ U_2 \end{pmatrix}^T \begin{pmatrix} W_2 \\ U_2 \end{pmatrix} = I,$$

then it follows that left factor

$$\begin{pmatrix} U_1 W_2 \\ U_2 \end{pmatrix}$$

also has orthonormal columns. If this is true always we say that the SSS representation is in **left proper form**. The rest of the construction process should be clear now for producing all the U_i's and W_i's.

Furthermore if

$$\operatorname{rank} \begin{pmatrix} A_{1,i+1} & A_{1,i+2} & \cdots \\ \vdots & \vdots & \\ A_{i-1,i+1} & A_{i-1,i+2} & \\ A_{i,i+1} & A_{i,i+2} & \cdots \end{pmatrix} = m_i$$

then it follows that W_i and U_i have m_i columns. As to the V_i's note that

$$V_{i+1}^T = H_{i,i+1}^T.$$

Therefore it also follows that V_{i+1} has exactly m_i columns.

A similar procedure (or work on A^T) can be used to construct P, Q, R. We will use n_i to denote the ranks of Hankel blocks with top right corner block $A_{i-1,i}$.

The construction procedure shows several things:

- Every matrix A has an exact SSS representation.

- If there are n block rows then the cost of constructing an SSS representation is $O(n^2\,m^2)$ flops, where $m = \max(m_i)$.
- The SSS representation compresses each Hankel block optimally.

The SSS representation enjoys further nice properties:

- It allows fast matrix vector multiplies, Ax.
- It allows fast solution of $Ax = b$.
    - It allows fast LU, QR and ULV factorizations of A.
- If both A and B are in SSS form:
    - it allows fast addition $A + B$, and
    - fast multiplication AB.

So the SSS form can be viewed as an ideal matrix representation for fast algorithms, provided of course the matrices have low Hankel ranks. (For further information on SSS representations see the book by Dewilde & van der Veen.)

Here we just cover fast matrix-vector multiply and fast solvers, as they are tied to each other.

14.4 Fast matrix-vector multiplication

Given x and A in SSS form we want to rapidly compute $b = Ax$. Looking at the i-th block component of b we get:

$$
\begin{aligned}
b_i &= \sum_{j=1}^{i-1} A_{i,j}\,x_j + A_{i,i}\,x_i + \sum_{j=i+1}^{n} A_{i,j}\,x_j \\
&= \sum_{j=1}^{i-1} P_i\,R_{i-1}\cdots R_{j+1}\,Q_j^T x_j + D_i\,x_i + \sum_{j=i+1}^{n} U_i\,W_{i+1}\cdots W_{j-1}\,V_j^T x_j \\
&= \underbrace{P_i \sum_{j=1}^{i-1} R_{i-1}\cdots R_{j+1}\,Q_j^T x_j}_{h_{i-1}} + D_i\,x_i + \underbrace{U_i \sum_{j=i+1}^{n} W_{i+1}\cdots W_{j-1}\,V_j^T x_j}_{g_{i+1}} \\
&= P_i\,h_{i-1} + D_i\,x_i + U_i\,g_{i+1} \\
&= P_i\left(R_{i-1}\left(\underbrace{\sum_{j=1}^{i-2} R_{i-2}\cdots R_{j+1}\,Q_j^T x_j}_{h_{i-2}} \right) + Q_{i-1}^T x_{i-1} \right) \\
&+ D_i\,x_i \\
&+ U_i\left(V_{i+1}^T\,x_{i+1} + W_{i+1}\left(\underbrace{\sum_{j=i+2}^{n} W_{i+2}\cdots W_{j-1} V_j^T x_j}_{g_{i+2}} \right) \right) \\
&= P_i(R_{i-1}\,h_{i-2} + Q_{i-1}^T x_{i-1}) + D_i\,x_i + U_i(V_{i+1}^T\,x_{i+1} + W_{i+1}\,g_{i+2}).
\end{aligned}
$$

This prompts use to define the following recursive procedure:

$$
\begin{aligned}
g_n &= V_n^T x_n \\
g_i &= V_i^T x_i + W_i\,g_{i+1} \\
h_1 &= Q_1^T x_1 \\
h_i &= Q_i^T x_i + R_i\,h_{i-1} \\
b_i &= P_i\,h_{i-1} + D_i\,x_i + U_i\,g_{i+1}.
\end{aligned}
$$

A simple count shows that the cost is $O(nm^2)$ flops, where $m = \max(m_i, n_i)$. If we want to use the matrix size N rather than the block size n, then this estimate is $O(Nm)$ flops.

14.5 Sparse representation and fast solver

We can now re-express the fast multiplication recursions in sparse "diagonal" form. Define the matrices

$$\begin{aligned} D &= \text{Diag}[D_i] \\ V &= \text{Diag}[V_i] \\ U &= \text{Diag}[U_i] \\ W &= \text{Diag}[W_i] \\ P &= \text{Diag}[P_i] \\ Q &= \text{Diag}[Q_i] \\ R &= \text{Diag}[R_i] \\ g &= \text{Vec}[g_i] = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \end{pmatrix} \\ h &= \text{Vec}[h_i] \\ x &= \text{Vec}[x_i] \\ b &= \text{Vec}[b_i]. \end{aligned}$$

We will assume that $Z_\downarrow$ denotes a suitable block down shift matrix and $Z_\uparrow$ a suitable block up shift matrix. Then the fast multiplication recursions can be expressed as:

$$\begin{pmatrix} I - WZ_\uparrow & 0 & -V^T & 0 \\ 0 & I - RZ_\downarrow & -Q^T & 0 \\ UZ_\uparrow & PZ_\downarrow & D & -I \end{pmatrix} \begin{pmatrix} g \\ h \\ x \\ b \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},$$

which is a very sparse matrix. We can express fast matrix-vector multiply as the triangular equation:

$$\begin{pmatrix} I - WZ_\uparrow & 0 & 0 \\ 0 & I - RZ_\downarrow & 0 \\ -UZ_\uparrow & -PZ_\downarrow & I \end{pmatrix} \begin{pmatrix} g \\ h \\ b \end{pmatrix} = \begin{pmatrix} V^T x \\ Q^T x \\ -Dx \end{pmatrix},$$

so fast matrix-vector is just sparse forward substitution. We make a useful observation. Note that

$$\begin{aligned} (I - WZ_\uparrow)^{-1} &= \begin{pmatrix} I & -W_2 & & & \\ & I & -W_3 & & \\ & & \ddots & \ddots & \\ & & & I & -W_{n-1} \\ & & & & I \end{pmatrix}^{-1} \\ &= \begin{pmatrix} I & W_2 & W_2W_3 & W_2W_3W_4 & \cdots \\ & I & W_3 & W_3W_4 & \cdots \\ & & I & W_4 & \cdots \\ & & & \ddots & \ddots \end{pmatrix}. \end{aligned}$$

Next for solving $Ax = b$ we can re-arrange the system suitably and first obtain:

$$\begin{pmatrix} I - WZ_\uparrow & 0 & -V^T \\ 0 & I - RZ_\downarrow & -Q^T \\ UZ_\uparrow & PZ_\downarrow & D \end{pmatrix} \begin{pmatrix} g \\ h \\ x \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ b \end{pmatrix}.$$

However, we see the dreaded bad ordering of the arrowhead matrix and sure enough doing sparse Gaussian elimination leads to fill-in, but yet yields useful information:

$$A = D + UZ_{\uparrow}(I - WZ_{\uparrow})^{-1}V^T + PZ_{\downarrow}(I - RZ_{\downarrow})^{-1}\,Q^T.$$

This is called the diagonal representation of A, however it does not yield a fast algorithm for the solution x. For that we must find an ordering of the rows and columns that minimizes fill-in. A good ordering can be obtained as follows:

$$y_i \;=\; \begin{pmatrix} g_i \\ h_i \\ x_i \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & -R_i & 0 \\ 0 & P_i & 0 \end{pmatrix} y_{i-1} + \begin{pmatrix} I & 0 & -V_i^T \\ 0 & I & -Q_i^T \\ 0 & 0 & D_i \end{pmatrix} y_i + \begin{pmatrix} -W_i & 0 & 0 \\ 0 & 0 & 0 \\ U_i & 0 & 0 \end{pmatrix} y_{i+1} \;=\; \begin{pmatrix} 0 \\ 0 \\ b_i \end{pmatrix}.$$

This is a banded system of equations and can be solved in $O(nm^3)$ flops.

While convenient and highly practical, this approach is not that satisfactory. For example, it does not readily provide the SSS representation of the LU factors of A, though some such form must have been computed. Fortunately there do exists direct fast algorithms for all of these operations, but we postpone them for now.

15 HSS

15.1 Introduction

The SSS representation is not as efficient in practice when there are Hankel blocks with large ranks. In particular for potential theory problems in 2 or 3 dimensions, and doubly Pick (Cauchy) matrices, SSS does not seem to capture all the structure of the matrix, so other representations have been investigated. Very likely the most appropriate one is the FMM one, but unfortunately a good ordering for the corresponding sparse matrix is not known, and probably does not exist. In this context I with my collaborators proposed the HSS representation, which is something like a 0.5D FMM representation. However it comes with the advantage that a fast solver is known, and in fact it is very much like SSS, in that a full suite of fast algorithms for various operations are available. We point out that it was known even earlier in the thesis of Paige Starr (Rokhlin's student) that the HSS like matrices have fast direct solvers. However the proposed algorithm was numerically stable and Ming Gu was the first to propose the fast ULV solvers as a potential avenue of attack for these problems.

Even with all this progress the question of fast direct solvers for general FMM matrices remains wide open.

15.2 Basic facts

We start with some facts about low-rank off-diagonal blocks of 3×3 partitions. Since

$$\begin{pmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} A_{22} & A_{21} & A_{23} \\ A_{12} & A_{11} & A_{13} \\ A_{32} & A_{31} & A_{33} \end{pmatrix}$$

we see that the row-Hankel block

$$\begin{pmatrix} A_{21} & A_{23} \end{pmatrix}$$

is a true Hankel block after block permutations. Similarly for the column-Hankel block

$$\begin{pmatrix} A_{12} \\ A_{32} \end{pmatrix}.$$

Therefore it follows that any low-rank in these blocks will also be preserved across matrix operations. We leave the detailed verification of this claim to the reader.

Just as SSS representations were designed to capture precisely the low-rank of all Hankel blocks, HSS representations are designed to capture the low rank of all row- and column-Hankel blocks. However, now because of the interleaving of the permutations we need to take care of some book keeping, which was simple enough in the SSS case that we didn't call it out separately.

A good way is to categorize the blocks recursively. We start with a 2×2 partition:

$$\begin{aligned} A_{0;} &= A \\ A_{0;} &= \begin{array}{cc|cc} & & n_{1;0} & n_{1;1} \\ \hline n_{1;0} & & A_{1;0,0} & A_{1;0,1} \\ n_{1;1} & & A_{1;1,0} & A_{1;1,1} \end{array}, \end{aligned}$$

where the index before the semi-colon reminds us how many times we have recursively partitioned A, and the size of each partition, $n_{1;0}$ and $n_{1;1}$, are fixed based on other considerations. We then proceed to split each block further:

$$A_{l-1;i,j} = \begin{array}{cc|cc} & & n_{l;2i} & n_{l;2i+1} \\ \hline n_{l;2i} & & A_{l;2i,2i} & A_{l;2i,2i+1} \\ n_{l;2i+1} & & A_{l;2i+1,2i} & A_{l;2i+1,2i+1} \end{array}.$$

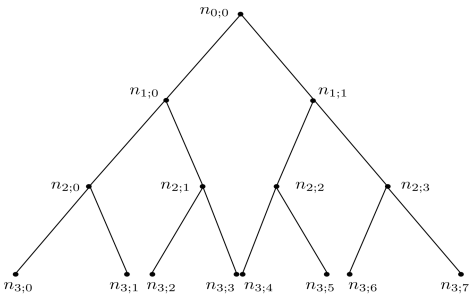
It is easier to understand the naming scheme by looking the full 2nd level partition

$$A = \begin{array}{cc|cccc} & & n_{2;0} & n_{2;1} & n_{2;2} & n_{2;3} \\ \hline n_{2;0} & & A_{2;0,0} & A_{2;0,1} & A_{2;0,2} & A_{2;0,3} \\ n_{2;1} & & A_{2;1,0} & A_{2;1,1} & A_{2;1,2} & A_{2;1,3} \\ n_{2;2} & & A_{2;2,0} & A_{2;2,1} & A_{2;2,2} & A_{2;2,3} \\ n_{2;3} & & A_{2;3,0} & A_{2;3,1} & A_{2;3,2} & A_{2;3,3} \end{array},$$

where the partitions from $A_{1;i,j}$ are shown in different colors. It is necessarily true that

$$n_{l;i} = n_{l+1;2i} + n_{l+1;2i+1}.$$

It helps to visualize these relationships using a binary partition tree:



Note that $A_{1;0,1}$ and $A_{1;1,0}$ are both Hankel blocks and it makes sense to posit that they have low rank expansions

$$A_{1;i,j} = U_{1;i} B_{1;i,j} V_{1;j}^T,$$

where, for stability purposes we will assume that both U_* and V_* have orthonormal columns. Now consider the sub-matrix $A_{2;0,1}$. It is a sub-matrix of the row-Hankel block

$$\begin{pmatrix} A_{2;0,1} & A_{2;0,2} & A_{2;0,3} \end{pmatrix}$$

and hence must have low rank, but it overlaps with $A_{1;0,1}$. Similarly $A_{2;0,1}$ is a sub-matrix of the column-Hankel block

$$\begin{pmatrix} A_{2;0,1} \\ A_{2;2,1} \\ A_{2;3,1} \end{pmatrix}$$

and hence must have low rank, but it overlaps with $A_{1;1,0}$. There are other such constraints at this level. So we need a representation that satisfies all these constraints optimally. The choice we make is that:

$$\begin{aligned} A_{2;i,j} &= U_{2;i}\, B_{2;i,j}\, V_{2;j}^T \\ U_{1;i} &= \begin{pmatrix} U_{2;2i}\, R_{2;2i} \\ U_{2;2i+1}\, R_{2;2i+1} \end{pmatrix} \\ V_{1;i} &= \begin{pmatrix} V_{2;2i}\, W_{2;2i} \\ V_{2;2i+1}\, W_{2;2i+1} \end{pmatrix}. \end{aligned}$$

It is conventional in HSS representations to only choose a minimal subset of these parameters to actually store and represent the matrix. The choice is dictated by a binary tree that the user can choose typically based on some other considerations. Here we will simply choose a simple uniform binary tree. For example the column-Hankel block

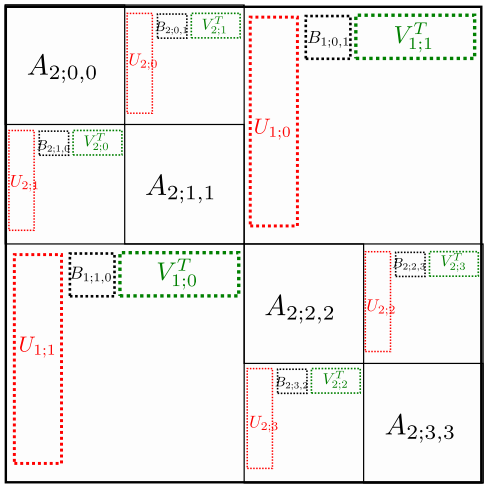
$$\begin{pmatrix} A_{2;0,1} & A_{2;0,2} \\ A_{2;3,1} & A_{2;3,2} \end{pmatrix}$$

will not be given an explicit representation. Instead they will obtain their representations from other row- and column-Hankel blocks. This choice is indicated by the coloring pattern below:

Singly colored blocks will have a stored representation of the form

$$U_{l;i}\, B_{l;i,j}\, V_{l;j}^T$$

where l depends on the smallest level at which the block can be first isolated. Representations of sub-blocks of singly colored blocks will not be stored or even computed as there is no need for them. Here is another visualization of what we mean:



Even these many parameters is too much. In particular note that because of $R_{l;i,j}$ and $W_{l;i,j}$, we do not need to store $U_{k;i}$ and $V_{k;i}$ except at the highest partition levels (when the blocks are smallest in size).

So, for example, if $l=2$ is the highest level partition, then the matrix representation will be as follows:

$$\begin{pmatrix} D_{2;0} & U_{2;0}\,B_{2;0,1}V_{2;1}^T & U_{2;0}R_{2;0}B_{1;0,1}W_{2;2}^TV_{2;2}^T & U_{2;0}R_{2;0}B_{1;0,1}W_{2;3}^TV_{2;3}^T \\ U_{2;1}\,B_{2;1,0}\,V_{2;0}^T & D_{2;1} & U_{2;1}R_{2;1}B_{1;0,1}W_{2;2}^TV_{2;2}^T & U_{2;0}R_{2;0}B_{1;0,1}W_{2;3}^TV_{2;3}^T \\ U_{2;2}R_{2;2}B_{1;1,0}W_{2;0}^TV_{2;0}^T & U_{2;2}R_{2;2}B_{1;1,0}W_{2;1}^TV_{2;1}^T & D_{2;2} & U_{2;2}\,B_{2;2,3}\,V_{2;3}^T \\ U_{2;3}R_{2;3}B_{1;1,0}W_{2;0}^TV_{2;0}^T & U_{2;3}R_{2;3}B_{1;1,0}W_{2;1}^TV_{2;1}^T & U_{2;3}\,B_{2;3,2}\,V_{2;2}^T & D_{2;3} \end{pmatrix}.$$

It is worthwhile to peruse this example to get a feel for what part of the representation is actually stored, and how precisely the different blocks are reconstructed. Just like the SSS case simple low-rank decomposition of the smallest blocks are not very useful.

The main goal of the HSS representation is to constuct fast direct solvers. If fast matrix-vector multiplication is desired then the full FMM representation will be a better choice. Other representations have been proposed, but they are typically sub-optimal for both operations, and we do not discuss them here.

15.3 Construction

Just like in the SSS case it is useful to start with the construction algorithm. We start with the construction of the $U_{k;i}$'s and their associated $R_{k;i}$'s. The same algorithm applied to the transpose will then yield the corresponding set of $V_{k;i}$'s and $W_{k;i}$'s. Once these are available the $B_{k;i,j}$'s can then be computed.

We assume that the desired binary partition tree is already given (presumably chosen so as to ensure good low-rank properties for all associated row- and column-Hankel blocks).

We start at the deepest level $l=L$ and look at each row-Hankel block at that level:

$$H_{l;i} = \begin{pmatrix} A_{l;i,0} & A_{l;i,1} & \cdots A_{l;i,i-1} & A_{l;i,i+1} & A_{l;i,i+2} & \cdots & A_{l;i,2^l-1} \end{pmatrix}.$$

We compute a low-rank left orthogonal factorization:

$$H_{l;i} = U_{l;i} \begin{pmatrix} G_{l;i,0} & \cdots G_{l;i,i-1} & G_{l;i,i+1} & \cdots & G_{l;i,2^l-1} \end{pmatrix}.$$

We now move one level down and consider the row-Hankel blocks:

$$\begin{aligned} H_{l-1;i} &= \begin{pmatrix} A_{l;2i,0} & \cdots & A_{l;2i,2i-1} & A_{l;2i,2i+2} & \cdots & A_{l;2i,2^l-1} \\ A_{l;2i+1,0} & \cdots & A_{l;2i+1,2i-1} & A_{l;2i,2i+2} & \cdots & A_{l;2i,2^l-1} \end{pmatrix} \\ &= \begin{pmatrix} U_{l;2i} & \\ & U_{l;2i+1} \end{pmatrix} \begin{pmatrix} G_{l;2i,0} & \cdots & G_{l;2i,2i-1} & G_{l;2i,2i+2} & \cdots & G_{l;2i,2^l-1} \\ G_{l;2i+1,0} & \cdots & G_{l;2i+1,2i-1} & G_{l;2i,2i+2} & \cdots & G_{l;2i,2^l-1} \end{pmatrix} \\ &= \begin{pmatrix} U_{l;2i} & \\ & U_{l;2i+1} \end{pmatrix} \begin{pmatrix} R_{l;2i} \\ R_{l;2i+1} \end{pmatrix} \begin{pmatrix} G_{l-1;i,0} & \cdots G_{l-1;i,i-1} & G_{l-1;i,i+1} & \cdots & G_{l-1;i,2^{l-1}-1} \end{pmatrix} \end{aligned}$$

where the last factorization is again assumed to produce orthonormal columns for the matrix

$$\begin{pmatrix} R_{l;2i} \\ R_{l;2i+1} \end{pmatrix}$$

to ensure numerical stability of the representation. Now the recursion proceeds up the partition tree in an obvious way. Note that at each stage we attain optimal compression of each row-Hankel block as promised. It is important pay attention to the fat matrices in the above recursions as we are carefully dropping diagonal blocks as we work our way up the partition tree.

We use a similar recursion on the column-Hankel blocks to compute the $V_{k;i}$'s and the $W_{k;i}$'s.

Now we turn our attention to the $B_{k;i,j}$'s. The most obvious approach is to re-use the compressed row-Hankel (say) blocks:

$$\begin{aligned} B_{l;2i,2i+1} &= U_{l;2i}^T A_{l;2i,2i+1} V_{l;2i+1}, \\ B_{l;2i+1,2i} &= U_{l;2i+1}^T A_{l;2i+1,2i} V_{l;2i}. \end{aligned}$$

At the highest level $l = L$ this is an efficient formula. At a lower level:

$$\begin{aligned} B_{l-1:i,j} &= U_{l-1;i}^T A_{l-1;i,j} V_{l-1,j} \\ &= \begin{pmatrix} R_{l;2i}^T U_{l;2i}^T & R_{l;2i+1}^T U_{l;2i+1}^T \end{pmatrix} \begin{pmatrix} A_{l;2i,2j} & A_{l;2i,2j+1} \\ A_{l;2i+1,2j} & A_{l;2i+1,2j+1} \end{pmatrix} \begin{pmatrix} V_{l;2j} W_{l;2j} \\ V_{l;2j+1} W_{l;2j+1} \end{pmatrix} \\ &= R_{l;2i}^T B_{l;2i,2j} W_{l;2j} + R_{l;2i+1}^T B_{l;2i+1,2j} W_{l;2j} + R_{l;2i}^T B_{l;2i,2j+1} W_{l;2j+1} + \\ &\quad + R_{l;2i+1}^T B_{l;2i+1,2j+1} W_{l;2j+1}. \end{aligned}$$

So this leads to a recursive construction for each local block that needs $B_{k;i,j}$.

It is easy to check that the total cost of this construction is $O(n^2 k)$ where k is the maximum Hankel (row and column) rank encountered.

We again note that if the matrix entries are special, for example $A_{i,j} = \sqrt{\|x_i - x_j\|}$, then FMM techniques can be used to compute the HSS representation at a cost determined by the dimension of the vector space containing the points x_i. We refer to the copious FMM literature for these issues.

15.4 Fast matrix-vector multiply

Suppose we have the HSS representation of A, given the column matrix x, how quickly can we compute $b = Ax$? The spirit of the algorithm is similar to the SSS case.

The key is to look at the inevitable product

$$A_{1;0,0} x_{1;0} + A_{1;0,1} x_{1;1} = b_{1;0}$$

where the partitions $x_{l;i}$ and $b_{l;i}$ are conformal with $n_{l;i}$. We see that this product involves:

$$A_{1;0,0} x_{1;0} + U_{1;0} B_{1;0,1} \underbrace{V_{1;1}^T x_{1;1}}_{g_{1;1}} = b_{1;0},$$

but we do not have $V_{1;1}$. So we must recurse down:

$$\begin{aligned} g_{1;1} &= V_{1;1}^T x_{1;1} \\ &= \begin{pmatrix} W_{2;2}^T V_{2;2}^T & W_{2;3}^T V_{2;3}^T \end{pmatrix} \begin{pmatrix} x_{2;2} \\ x_{2;3} \end{pmatrix} \\ &= W_{2;2}^T g_{2;2} + W_{2;3}^T x_{2;3}. \end{aligned}$$

We see now that these quantities are needed and prompts us to write down the general bottom-up recursions on the partition tree:

$$\begin{aligned} g_{L;i} &= V_{L;i}^T x_{L;i} \\ g_{l-1;i} &= W_{l;2i}^T g_{l;2i} + W_{l;2i+1}^T g_{l;2i+1}. \end{aligned}$$

At the end of this recursion we basically have all the $V_{l;i}^T x_{l;i}$ that we could possibly need. Now we go back to the rest of the terms for $b_{1;0}$:

$$A_{1;0,0}\, x_{1;0} + U_{1;0} \underbrace{B_{1;0,1}\, g_{1;1}}_{h_{1;0}} = b_{1;0}.$$

We do not have $U_{1;0}$, so we must recurse:

$$\begin{aligned}
b_{1;0} &= A_{1;0,0}\, x_{1;0} + U_{1;0}\, h_{1;0} \\
\begin{pmatrix} b_{2;0} \\ b_{2;1} \end{pmatrix} &= \begin{pmatrix} A_{2;0,0} & U_{2;0}\, B_{2;0,1}\, V_{2;1}^T \\ U_{2;1}\, B_{2;1,0}\, V_{2;0}^T & A_{2;1,1} \end{pmatrix} \begin{pmatrix} x_{2;0} \\ x_{2;1} \end{pmatrix} + \begin{pmatrix} U_{2;0}\, R_{2;0} \\ U_{2;1}\, R_{2;1} \end{pmatrix} h_{1;0} \\
b_{2;0} &= A_{2;0,0}\, x_{2;0} + U_{2;0} \left(\underbrace{B_{2;0,1}\, g_{2;1} + R_{2;0}\, h_{1;0}}_{h_{2;0}} \right) \\
b_{2;1} &= A_{2;1,1}\, x_{2;1} + U_{2;1} \left(\underbrace{B_{2;1,0}\, g_{2;0} + R_{2;1}\, h_{1;0}}_{h_{2;1}} \right).
\end{aligned}$$

So examining these formulas we see that we must use the general recursions:

$$\begin{aligned}
h_{0;0} &= [] \\
h_{l;2i} &= B_{l;2i,2i+1}\, g_{l;2i+1} + R_{l;2i}\, h_{l-1;i} \\
h_{l;2i+1} &= B_{l;2i+1,2i}\, g_{l;2i} + R_{l;2i+1}\, h_{l-1;i}.
\end{aligned}$$

Finally the above expressions also give us:

$$b_{L;i} = A_{L;i,i}\, x_{L;i} + U_{L;i}\, h_{L;i}.$$

It is easy to check that the cost of this algorithm is $O(Nk)$ flops, where k is the maximum Hankel rank.

15.5 Fast solver

We will again take the path of representing the fast multiplication algorithm as a sparse system of equations. We will organize the intermediate variables in a breadth-first manner:

$$\begin{aligned}
g_l &= \begin{pmatrix} g_{l;0} \\ g_{l;1} \\ \vdots \\ g_{l;2^l-1} \end{pmatrix} \\
g &= \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_L \end{pmatrix}.
\end{aligned}$$

Similarly for h.