An introduction to recent fast algorithms for structured matrices

Shivkumar Chandrasekaran
shiv@ucsb.edu

Nithin Govindarajan
ngovindarajan@ucsb.edu

Kristen Lessel
klessel@ucsb.edu

Abhejit Rajagopal
abhejit@ucsb.edu

July 1, 2018

Introduction

We will be primarily interested in the following questions.

- If A is a real square n-by-n matrix ($A \in \mathbb{R}^{n \times n}$) and x is a column vector ($x \in \mathbb{R}^n$) how quickly can we compute:
  - Ax?
  - $A^{-1}x$?
- The naive algorithms for the above two problems require $O(n^2)$ and $O(n^3)$ operations respectively.
- Strassen-style [9] algorithms can improve the latter to $O(n^{2.3\cdots})$ but are impractical.
- If there is more *structure* in the matrix A, can we do better?

Outline of the Talk

Examples of structured matrices

Strassen-style algorithms

FFT and Toeplitz

Fast classical polynomial arithmetic

Displacement structure

Low rank structure

Sequentially Semi-Separable Representations

Hierarchically semi-separable representations

Concluding remarks

Thank you

Examples of structured matrices

Toeplitz & Vandermonde
Cauchy & Banded
Generalizations
More questions

Toeplitz & Vandermonde

- Toeplitz matrices:

$$\begin{bmatrix} t_0 & t_1 & t_2 & \cdots & & \\ t_{-1} & t_0 & t_1 & t_2 & \cdots & \\ t_{-2} & t_{-1} & t_0 & t_1 & t_2 & \cdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}$$

- Vandermonde matrices:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & \cdots \\ 1 & x_1 & x_1^2 & x_1^3 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

Cauchy & Banded

- Cauchy matrices:

$$\begin{bmatrix} \frac{1}{x_0-y_0} & \frac{1}{x_0-y_1} & \frac{1}{x_0-y_2} & \cdots \\ \frac{1}{x_1-y_0} & \frac{1}{x_1-y_1} & \frac{1}{x_1-y_2} & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix}$$

- Banded matrices:

$$\begin{bmatrix} a_0 & b_0 & & & \\ c_0 & a_1 & b_1 & & \\ & c_1 & a_2 & \ddots & \\ & & \ddots & \ddots & \ddots \end{bmatrix}$$

Generalizations

- Blocked versions.
- Inverses of structured matrices.
- Generalizations of Cauchy matrices where the kernel $f(x, y) = (x - y)^{-1}$, is replaced by functions $f : \mathbb{R}^d \to \mathbb{R}$ with structured singularities. Example,

$$f(x, y) = \log \|x - y\|.$$

- Sparse matrices induced from graphs.
- Matrices of the type:

$$A + BC^{-1}D$$

  where A, B, C and D are all structured matrices.
- Sums of tensor products of structured matrices.

More questions

- Examples required $O(n)$ numbers to represent $n \times n$ matrix.
- Can we get $O(n \log^k n)$ algorithms for Ax and $A^{-1}x$ computations?
- Can we get $O(n \log^k n)$ algorithms for Gaussian elimination (LU), Gram–Schmidt (QR), singular value decomposition ($U\Sigma V^T$) and eigendecomposition ($V\Lambda V^{-1}$)?
- Are the algorithms numerically stable?
- If not can we get slower, say $O(n^2)$ for $A^{-1}x$, that are numerically stable?

Answers are known only in special cases.

Strassen-style algorithms

Fast matrix multiplication
Trading multiplies for additions
Strassen-type formulas

Fast matrix multiplication

Consider the 2×2 block product of 2 matrices:

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

with

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

This requires

- 8 block multiplies: $O(n^3)$ operations
- 4 block adds: $O(n^2)$ operations

Strassen's idea is to trade matrix multiplies for additions, which are far cheaper, and the use the formula **recursively**.

Trading multiplies for additions

The right way to look at Strassen is to study the bilinear operator $(A, B) \to AB$.

Consider $P \times Q$ and $Q \times R$ block equi-partitions of A and B respectively. For some L, suppose

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

and

$$\sum_{k \in Q} A_{i,k} B_{k,j} = \sum_{l \in L} \lambda_{i,j}^l M_l, \qquad i = 1, \ldots, P, \qquad j = 1, \ldots, R,$$

for scalars $\alpha_{i,j}^l$, $\beta_{i,j}^l$ and $\lambda_{i,j}^l$.

Strassen-type formulas

- If we can find scalars $\alpha_{i,j}^l$, $\beta_{i,j}^l$ and $\lambda_{i,j}^l$ such that these equations hold then we would have a formula with $O(Q(P+R)L)$ additions and L multiplies.
- To be fast we need $L < PQR$.
- There are $(Q(P+R)+PR)L$ variables and $(PQR)^2$ tri-linear equations.
- Strassen's first solution was with $P = Q = R = 2$ and $L = 7$.
- Finding P, Q, R and L such that there is a solution is non-trivial.
- The larger P, Q and R are the less practical would the method be.

We will not pay much attention to Strassen-style acceleration in this tutorial.

FFT and Toeplitz

- Discrete Fourier series
- FFT
- Circulant matrices
- FFT diagonalizes circulants
- Padding
- Toeplitz multiplication
- Any sized FFT
- Fast Toeplitz back substitution

Discrete Fourier series

For positive integer n let $\omega_n = \exp\left(\iota\frac{2\pi}{n}\right)$, with $\iota^2 = -1$.
The discrete $n \times n$ Fourier matrix is defined to be:

$$F_n = \frac{1}{\sqrt{n}}\begin{bmatrix} \omega_n^{0\cdot 0} & \omega_n^{0\cdot 1} & \omega_n^{0\cdot 2} & \dots & \omega_n^{0\cdot(n-1)} \\ \omega_n^{1\cdot 0} & \omega_n^{1\cdot 1} & \omega_n^{1\cdot 2} & \dots & \omega_n^{1\cdot(n-1)} \\ \omega_n^{2\cdot 0} & \omega_n^{2\cdot 1} & \omega_n^{2\cdot 2} & \dots & \omega_n^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ \omega_n^{(n-1)\cdot 0} & \omega_n^{(n-1)\cdot 1} & \omega_n^{(n-1)\cdot 2} & \dots & \omega_n^{(n-1)\cdot(n-1)} \end{bmatrix}$$

Let

$$\Omega_n = \operatorname{diag}\begin{bmatrix} \omega_{2n}^0 & \omega_{2n}^1 & \omega_{2n}^2 & \cdots & \omega_{2n}^{n-1} \end{bmatrix}$$

FFT

It is easy to check that:

$$F_{2n} = \frac{1}{\sqrt{2}} \begin{bmatrix} I & \Omega_n \\ I & -\Omega_n \end{bmatrix} \begin{bmatrix} F_n & 0 \\ 0 & F_n \end{bmatrix}$$

- The **recursive** application of this idea, when $n = 2^k$, enables the computation of $F_n x$ in $O(n \log_2 n)$ operations. This is called the FFT.
- Note that $F_n^{-1} = F_n^H$, so $F_n^{-1} x$ can also be computed in $O(n \log_2 n)$ operations.
- The case $n \neq 2^k$ will be covered later.
- The FFT [4] is the gateway to many classical fast algorithms.

Circulant matrices

$$C(c) = \begin{bmatrix} c_0 & c_{n-1} & \cdots & \cdots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & \cdots & \cdots & c_2 \\ \vdots & c_1 & \ddots & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & c_{n-1} & \vdots \\ c_{n-2} & \vdots & & c_1 & c_0 & c_{n-1} \\ c_{n-1} & c_{n-2} & \cdots & \cdots & c_1 & c_0 \end{bmatrix}$$

The circular shift down matrix:

$$Z_{\circlearrowleft} = \begin{bmatrix} 0 & \cdots & \cdots & 0 & 1 \\ 1 & 0 & & \vdots & 0 \\ 0 & 1 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix} = C(e_1).$$

FFT diagonalizes circulants

Let

$$\Lambda_n = \operatorname{diag}\begin{bmatrix}\omega_n^0 & \omega_n^1 & \omega_n^2 & \cdots & \omega_n^{n-1}\end{bmatrix}$$

Easy to check that

$$Z_{\circlearrowleft} = F_n \Lambda_n F_n^H$$

Since

$$C(c) = \sum_{k=0}^{n-1} c_k \, Z_{\circlearrowleft}^k$$

it is easy to check that

$$C(c) = F_n \operatorname{diag}\left(\sqrt{n} F_n c\right) F_n^H$$

Therefore if $c \in \mathbb{R}^{2^k}$ we have $O(n \log_2 n)$ algorithms for computing $C(c)x$ and $C^{-1}(c)x$.

Padding

If the length of c is not an exact power of 2 we can pad it out. Here is a 3×3 example of $C(c)x = y$:

$$\begin{bmatrix} c_0 & c_2 & c_1 & 0 & 0 & 0 & c_2 & c_1 \\ c_1 & c_0 & c_2 & c_1 & 0 & 0 & 0 & c_2 \\ c_2 & c_1 & c_0 & c_2 & c_1 & 0 & 0 & 0 \\ 0 & c_2 & c_1 & c_0 & c_2 & c_1 & 0 & 0 \\ 0 & 0 & c_2 & c_1 & c_0 & c_2 & c_1 & 0 \\ 0 & 0 & 0 & c_2 & c_1 & c_0 & c_2 & c_1 \\ c_1 & 0 & 0 & 0 & c_2 & c_1 & c_0 & c_2 \\ c_2 & c_1 & 0 & 0 & 0 & c_2 & c_1 & c_0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \star \\ \star \\ \star \\ \star \\ \star \end{bmatrix}$$

Therefore **any** sized circulant matrix can be multiplied in $O(n \log_2 n)$ operations using powers-of-2 FFTs.

Toeplitz multiplication

A Toeplitz matrix can be embedded in a circulant matrix. Here is a 3×3 example:

$$\begin{bmatrix} t_0 & t_1 & t_2 & t_{-2} & t_{-1} \\ t_{-1} & t_0 & t_1 & t_2 & t_{-2} \\ t_{-2} & t_{-1} & t_0 & t_1 & t_2 \\ t_2 & t_{-2} & t_{-1} & t_0 & t_1 \\ t_1 & t_2 & t_{-2} & t_{-1} & t_0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \star \\ \star \end{bmatrix}$$

- Therefore we can compute Tx for **any** sized Toeplitz matrix T in $O(n \log_2 n)$ operations.
- Unfortunately this does **not** yield a technique for the fast computation of $T^{-1}x$.

Any sized FFT

Let

$$\Gamma_n = \operatorname{diag}\begin{bmatrix} \exp\left(-\iota\frac{\pi}{n}0^2\right) \\ \exp\left(-\iota\frac{\pi}{n}1^2\right) \\ \cdots \\ \exp\left(-\iota\frac{\pi}{n}(n-1)^2\right) \end{bmatrix}$$

It is easy to check that $\Gamma_n F_n \Gamma_n$ is the Toeplitz matrix:

$$\begin{bmatrix} \exp\left(-\iota\frac{\pi}{n}0^2\right) & \exp\left(-\iota\frac{\pi}{n}1^2\right) & \cdots & \exp\left(-\iota\frac{\pi}{n}(n-1)^2\right) \\ \exp\left(-\iota\frac{\pi}{n}1^2\right) & \exp\left(-\iota\frac{\pi}{n}0^2\right) & \cdots & \exp\left(-\iota\frac{\pi}{n}(n-2)^2\right) \\ \vdots & \vdots & & \vdots \\ \exp\left(-\iota\frac{\pi}{n}(n-1)^2\right) & \exp\left(-\iota\frac{\pi}{n}(n-2)^2\right) & \cdots & \exp\left(-\iota\frac{\pi}{n}0^2\right) \end{bmatrix}$$

Therefore we can compute **any** sized $F_n x$ and $F_n^H x$ in $O(n \log_2 n)$ operations.

Fast Toeplitz back substitution

If T is a upper triangular Toeplitz matrix then $T^{-1}x$ can be computed in $O(n \log_2^2 n)$ operations using the classical "divide and conquer" approach.

- $$T^{-1}x = \begin{bmatrix} T_0 & T_1 \\ 0 & T_2 \end{bmatrix}^{-1} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} T_0^{-1}(x_0 - T_1(T_2^{-1}x_1)) \\ T_2^{-1}x_1 \end{bmatrix}$$

- Since all T_i are Toeplitz we can apply this recursively to compute $T^{-1}x$. The number of operations $f(\cdot)$ satisfies the recurrence

$$f(n) \leq 2f\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \log_2 n$$

for some constant c.

Fast classical polynomial arithmetic

- Multiplication
- Division
- Division
- Expansion of products
- Evaluation
- Lagrange sums
- Interpolation

Multiplication

We will restrict ourselves to monomial basis for the nonce.

$$\left(\sum_i a_i x^i\right)\left(\sum_j b_j x^j\right) = \sum_k c_k x^k$$

can be written as

$$\begin{bmatrix} a_0 & 0 & \cdots & & \\ a_1 & a_0 & \ddots & & \\ a_2 & a_1 & a_0 & \ddots & \\ \vdots & \ddots & \ddots & \ddots & \ddots \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{bmatrix}$$

This yields an $O(n \log_2 n)$ algorithm.

Division

Given

- polynomial $c(x)$ of degree n
- polynomial $a(x)$ of degree p

Find

- polynomial $q(x)$ of degree $n - p$
- polynomial $r(x)$ of degree $p - 1$

such that $c(x) = a(x)q(x) + r(x)$. Expanding in the monomial basis:

$$\left(\sum_{i=0}^{p} a_n x^n\right)\left(\sum_{j=0}^{n-p} q_j x^j\right) + \sum_{j=0}^{p-1} r_j x^j = \sum_{k=0}^{n} c_k x^k.$$

Division

In matrix form we have:

$$\left[\begin{array}{cccccccc}
a_0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
a_1 & a_0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
a_{p-1} & a_{p-2} & \cdots & a_0 & 0 & 0 & \cdots & 0 \\
\hline
a_p & a_{p-1} & \cdots & a_1 & a_0 & 0 & \cdots & 0 \\
0 & a_p & \cdots & a_2 & a_1 & a_0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & a_p & a_{p-1} & a_{p-2} & \cdots & a_0 \\
0 & 0 & \cdots & 0 & a_p & a_{p-1} & \cdots & a_1 \\
0 & 0 & \cdots & 0 & 0 & a_p & \cdots & a_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & a_p
\end{array}\right]
\left[\begin{array}{c} q_0 \\ q_1 \\ \vdots \\ q_{p-1} \\ q_p \\ q_{p+1} \\ \vdots \\ q_{n-p} \end{array}\right]
+
\left[\begin{array}{c} r_0 \\ r_1 \\ \vdots \\ r_{p-1} \\ \hline 0 \\ 0 \\ \vdots \\ 0 \end{array}\right]
=
\left[\begin{array}{c} c_0 \\ c_1 \\ \vdots \\ c_{p-1} \\ \hline c_p \\ c_{p+1} \\ \vdots \\ c_n \end{array}\right]$$

- The q_i's are found in $O(n \log_2^2 n)$ operations.
- The r_i's are found in $O(n \log_2 n)$ operations.

Expansion of products

Given complex numbers x_0, x_1, $\cdots$, x_{n-1} define

$$q_{r:s}(x) = \Pi_{i=r}^{s}(x - x_i)$$

Note that

$$q_{0:n-1} = q_{0:(n\div 2)} q_{(n\div 2+1):n-1}$$

- Therefore by using this recursively we can compute the coefficients of $q_{0:n-1}$ in $O(n \log_2^2 n)$ operations.
- Note also that we compute the coefficients of $q_{0:(n\div 2)}$ and $q_{(n\div 2+1):n-1)}$ and many more such polynomials along the way. This is exploited in the next couple of slides.

Evaluation

Re-using x_i and $q_{r:s}$ from previous slide.
Given polynomial p of degree n, find $p(x_i)$ for $i = 0, \ldots, n-1$.
Let

$$\begin{aligned} p(x) &= q_{0:(n\div 2)}(x)\, d_0(x) + p_0(x) \\ p(x) &= q_{(n\div 2+1):n-1}(x)\, d_1(x) + p_1(x) \end{aligned}$$

Then note that

$$\begin{aligned} p(x_i) &= p_1(x_i), \qquad i = 0, \ldots, n \div 2, \\ p(x_i) &= p_2(x_i), \qquad i = (n \div 2) + 1, \ldots, n-1, \end{aligned}$$

and the degree of p_i is roughly $n/2$. Therefore by recursively applying this idea we can finish the evaluation in $O(n \log_2^3 n)$ operations.

Lagrange sums

Let x_i be as before. Let

$$q_{r:t:s} = (x - x_r)\cdots(x - x_{t-1})(x - x_{t+1})\cdots(x - x_s)$$

Let $p(x) = \sum_i a_i\, q_{0;i;n-1}(x)$. Note that

$$\begin{aligned} q_{0:i:n-1}(x) &= q_{0:i:(n\div 2)}(x)\, q_{(n\div 2+1):n-1}(x), && i = 0, \ldots, n \div 2, \\ q_{0:i:n-1}(x) &= q_{0:(n\div 2)}(x)\, q_{(n\div 2+1):i:n-1}(x), && i = (n \div 2) + 1, \ldots, n - 1. \end{aligned}$$

It follows that we can split p into two Lagrange sums p_0 and p_1:

$$p(x) = p_0(x)\, q_{(n\div 2+1):n-1}(x) + p_1(x)\, q_{0:(n\div 2)}(x)$$

Using this idea recursively we can compute the coefficients of p in $O(n \log_2^2 n)$ operations.

The famous Lagrange formula

$$p(x) = \sum_{i=0}^{n-1} \frac{p(x_i)}{q_{0;i;n-1}(x_i)} q_{0:i:n-1}(x)$$

expresses the polynomial p in terms of its values at the points x_i.
Note that

$$q_{0;i;n-1}(x_i) = q'_{0:n-1}(x_i).$$

We can compute the derivative of a polynomial in $O(n)$ operations.
Therefore the coefficients of the Lagrange interpolating formula can be computed in $O(n \log_2^3 n)$ operations.

Displacement structure

- Historical context
- Toeplitz
- Vandermonde
- Cauchy and Pick
- Displacement operator
- Similarity transformations
- Recovery
- Displacement rank of inverse
- Gaussian elimination
- Schur algorithm
- Toeplitz to Cauchy
- Diagonal plus rank one
- Inverse
- Products

Historical context

- Polynomial division can generate large coefficients, so most related algorithms are impractical (either unstable or memory hogs).
- Classical fast and super-fast Toeplitz algorithms are also unstable.
- The development of displacement rank by Kailath et. al. [8] finally lead to developments that helped resolve some of these issues.

Toeplitz

The basic idea is best explained with an example. Let

$$Z_{\downarrow} = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & \ddots & & \vdots \\ 0 & 1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

denote the down shift matrix. Then observe that

$$\begin{bmatrix} a & b & c \\ d & a & b \\ e & d & a \end{bmatrix} - Z_{\downarrow} \begin{bmatrix} a & b & c \\ d & a & b \\ e & d & a \end{bmatrix} Z_{\downarrow}^{T} = \begin{bmatrix} a & b & c \\ d & 0 & 0 \\ e & 0 & 0 \end{bmatrix}$$

has rank 2.

Vandermonde

Let

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots \\ 1 & x_1 & x_1^2 & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix}, \qquad D = \operatorname{diag}(x).$$

Observe that

$$V - DVZ_{\downarrow}^T = \begin{bmatrix} 1 & 0 & \cdots \\ 1 & 0 & \cdots \\ \vdots & \vdots & \end{bmatrix}$$

has rank 1.

Cauchy and Pick

Let

$$C = \begin{bmatrix} (x_0 - y_0)^{-1} & (x_0 - y_1)^{-1} & \cdots \\ (x_1 - y_0)^{-1} & (x_1 - y_1)^{-1} & \cdots \\ \vdots & \vdots & \end{bmatrix},$$

$D_x = \operatorname{diag}(x)$, and $D_y = \operatorname{diag}(y)$. Observe that

$$C - D_x^{-1} C D_y = D_x^{-1} \begin{bmatrix} 1 & 1 & \cdots \\ 1 & 1 & \cdots \\ \vdots & \vdots & \end{bmatrix}$$

has rank 1.

Displacement operator

- Motivated by the Toeplitz case we call

$$\mathcal{L}_{A,B}(T) = T - ATB^T = PQ^T$$

  the **displacement** equation for T, and (P, Q) the **generator** for T with $P, Q \in \mathbb{R}^{n \times r}$.
- The rank r of $\mathcal{L}_{A,B}(T)$ is called the **displacement rank** of T with respect to $\mathcal{L}_{A,B}$.

If $\mathcal{L}_{A,B}^{-1}(xy^T)$ can be computed rapidly when $x, y \in \mathbb{R}^n$, then we could use (P, Q) as the representation of T and hope to obtain fast algorithms. This pans out for Gaussian elimination.

Similarity transformations

- To invert $\mathcal{L}_{A,B}$ we need to make A and B triangular. This can be done via a similarity transformation
  $W^{-1}TV^{-1} - (W^{-1}AW)(W^{-1}TV^{-1})(VB^TV^{-1}) = (W^{-1}P)(Q^TV^{-1})$.
- For computational efficiency this requires that the Schur type decomposition of A and B can be computed rapidly and the corresponding W and V can be applied rapidly to P and Q respectively.
- Note that we do **not** need to compute $W^{-1}TV^{-1}$.
- From now on we will assume that A and B are already **triangular**.
- Note that this puts a major crimp on the class to which A and B can originally belong. All our examples so far are fine though.

Recovery

Let

$$T = \begin{bmatrix} \tau & t^T \\ s & T_1 \end{bmatrix} \qquad A = \begin{bmatrix} \alpha & 0 \\ b & A_1 \end{bmatrix} \qquad B = \begin{bmatrix} \beta & 0 \\ h & B_1 \end{bmatrix}$$

$$P = \begin{bmatrix} p^T \\ P_1 \end{bmatrix} \qquad Q = \begin{bmatrix} q^T \\ Q_1 \end{bmatrix}$$

From the displacement equation we obtain

$$\begin{aligned} \tau &= \frac{p^T q}{1 - \alpha\beta} \\ s &= (I - \beta A_1)^{-1}(P_1 q + \tau\beta b) \\ t &= (I - \alpha B_1)^{-1}(Q_1 p + \tau\alpha h) \end{aligned}$$

To recover T efficiently from (P, Q) we must be able to implement above formulas in $O(n)$ operations.

Displacement rank of inverse

Claim:

$$\mathsf{rank}(\mathcal{L}_{A,B}(T)) = \mathsf{rank}(\mathcal{L}_{A,B}(T^{-T})).$$

Follows from the identities

$$\begin{aligned}
\begin{bmatrix} T^{-1} & B^T \\ A & T \end{bmatrix} &= \begin{bmatrix} I & 0 \\ AT & I \end{bmatrix} \begin{bmatrix} T^{-1} & B^T \\ 0 & T - ATB^T \end{bmatrix} \\
&= \begin{bmatrix} I & B^T T^{-1} \\ 0 & I \end{bmatrix} \begin{bmatrix} T^{-1} - B^T T^{-1} A & 0 \\ A & T \end{bmatrix}
\end{aligned}$$

Natural question is whether we can go quickly from the generators for T to that of T^{-1}.
Answer: **Schur** algorithm.

Gaussian elimination (1/3)

One step of (two-sided) block Gaussian elimination on T:

$$\begin{bmatrix} 1 & 0 \\ -\frac{s}{\tau} & I \end{bmatrix} \begin{bmatrix} \tau & t^T \\ s & T_1 \end{bmatrix} \begin{bmatrix} 1 & -\frac{t^T}{\tau} \\ 0 & I \end{bmatrix} = \begin{bmatrix} \tau & 0 \\ 0 & T_1 - \frac{st^T}{\tau} \end{bmatrix} = \begin{bmatrix} \tau & 0 \\ 0 & S \end{bmatrix}$$

Let

$$W = \begin{bmatrix} 1 & 0 \\ \frac{s}{\tau} & I \end{bmatrix} \qquad V = \begin{bmatrix} 1 & \frac{t^T}{\tau} \\ 0 & I \end{bmatrix}$$

Then we can find $W^{-1}AW$ and $V^{-T}BV^T$:

$$\begin{bmatrix} 1 & 0 \\ -\frac{s}{\tau} & I \end{bmatrix} \begin{bmatrix} \alpha & 0 \\ b & A_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{s}{\tau} & I \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ b + (A_1 - \alpha I)\frac{s}{\tau} & A_1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ -\frac{t}{\tau} & I \end{bmatrix} \begin{bmatrix} \beta & 0 \\ h & B_1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{t}{\tau} & I \end{bmatrix} = \begin{bmatrix} \beta & 0 \\ h + (B_1 - \beta I)\frac{t}{\tau} & B_1 \end{bmatrix}$$

Gaussian elimination (2/3)

Let

$$b_1 = b + (A_1 - \alpha I)\frac{s}{\tau} \qquad h_1 = h + (B_1 - \beta I)\frac{t}{\tau}$$

then $(W^{-1}AW)(W^{-1}TV^{-1})(VB^TV^{-1})$ can be computed as:

$$\begin{bmatrix} \alpha & 0 \\ b_1 & A_1 \end{bmatrix}\begin{bmatrix} \tau & 0 \\ 0 & S \end{bmatrix}\begin{bmatrix} \beta & h_1^T \\ 0 & B_1^T \end{bmatrix} = \begin{bmatrix} \alpha\tau\beta & \alpha\tau h_1^T \\ b_1\tau\beta & A_1SB_1^T + b_1\tau h_1^T \end{bmatrix}$$

$W^{-1}P$:

$$\begin{bmatrix} 1 & 0 \\ -\frac{s}{\tau} & I \end{bmatrix}\begin{bmatrix} p^T \\ P_1 \end{bmatrix} = \begin{bmatrix} p^T \\ P_1 - \frac{sp^T}{\tau} \end{bmatrix}$$

$V^{-T}Q$:

$$\begin{bmatrix} 1 & 0 \\ -\frac{t}{\tau} & I \end{bmatrix}\begin{bmatrix} q^T \\ Q_1 \end{bmatrix} = \begin{bmatrix} q^T \\ Q_1 - \frac{tq^T}{\tau} \end{bmatrix}$$

Gaussian elimination (3/3)

Claim:

$$\mathsf{rank}(\mathcal{L}_{A_1,B_1}(S)) = \mathsf{rank}(\mathcal{L}_{A,B}(T)).$$

Displacement equation for Schur complement S

$$\begin{aligned} S - A_1 S B_1^T &= P_2 Q_2^T + b_1 \tau h_1^T \\ &= P_2 \left(I + \frac{q p^T}{\tau \alpha \beta} \right) Q_2^T \end{aligned}$$

using the recovery formulas for the transformed equation for T. If $\alpha\beta = 0$ it is easy to show that $\mathsf{rank}(P_2 Q_2^T) < r$.

Schur algorithm (1/2)

- The recursive application of this idea to S is called the Schur algorithm (originally discovered in complex analysis).
- Requires that A and B are special lower triangular.
  - Ax can be computed in $O(n)$ operations.
  - $(I - \beta A)^{-1}x$ can be computed in $O(n)$ operations
- The recovery formula computes the first row and column of T (and S) in $O(n)$ operations. This is enough to compute the LU factorization recursively.
- Hence the LU factorization of T can be computed in $O(n^2)$ operations (versus the slow $O(n^3)$ operations).

Schur algorithm (2/2)

The Schur algorithm is in general unstable.

- If A and B are diagonal then we can incorporate pivoting and obtain a fast $O(n^2)$ algorithm that is very stable [6].
- The general Schur algorithm can also be stabilized to some extent [3].
- If A and B are not diagonalizable carefully chosen low-rank perturbations can be used to fix the problem.
- The Schur algorithm produces **unstructured** L and U factors.

Toeplitz to Cauchy

Let:

$$Z_\phi = \begin{bmatrix} 0 & \cdots & \cdots & 0 & \phi \\ 1 & 0 & & \vdots & 0 \\ 0 & 1 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

Note: Eigendecomposition of Z_ϕ is a diagonal scaling of $Z_{\circlearrowleft}$'s decomposition.

- rank$(\mathcal{L}_{Z_\phi, Z_\phi}(T))$ is at most three if T is Toeplitz.
- Therefore a fast trigonometric transform will convert a Toeplitz matrix to a generalized Cauchy matrix.

Diagonal plus rank one

- If $\mathcal{L}_{A,B}$ is non-invertible (standard Cauchy matrix), we can fix it by considering $\mathcal{L}_{A+xy^T,B}$ for suitable $x, y \in \mathbb{R}^n$.
- If A is diagonal we can choose the eigenvector matrix of $A + xx^T$ to be a Cauchy matrix with orthogonal columns and this can be computed in $O(n^2)$ operations plus the cost of finding n roots of a secular equation [Gu & Eisenstat].

Inverse (1/2)

We can use the Schur algorithm to find the generators of the inverse. Note that T^{-1} is a Schur complement of

$$\begin{bmatrix} T & -I \\ I & 0 \end{bmatrix}$$

and the rank of

$$\begin{bmatrix} T & -I \\ I & 0 \end{bmatrix} - \begin{bmatrix} A & 0 \\ 0 & B^{T\dagger} \end{bmatrix} \begin{bmatrix} T & -I \\ I & 0 \end{bmatrix} \begin{bmatrix} B^T & 0 \\ 0 & A^\dagger \end{bmatrix}$$

is at most

$$\mathsf{rank}(\mathcal{L}_{A,B}(T)) + \mathsf{rank}(I - AA^\dagger) + \mathsf{rank}(I - BB^\dagger)$$

Inverse (2/2)

- By running the Schur algorithm half-way through, the generators for the inverse can be computed in $O(n^2)$ operations
- Applied to a Toeplitz matrix this yields the Gohberg–Semencul formula for the inverse of Toeplitz matrices.

Products

Claim: Rank of

$$\mathcal{L}_{A_1,B_2}(T_1T_2)$$

is at most

$$\mathsf{rank}(\mathcal{L}_{A_1,B_1}(T_1)) + \mathsf{rank}(\mathcal{L}_{A_2,B_2}(T_2)) + \mathsf{rank}(I - B_1^T A_2)$$

Proof is an easy calculation.

- The generators for T_1T_2 can be computed in $O(n^2)$ operations.
- Product of two Toeplitz-like matrices is Toeplitz-like with larger displacement rank.
- Product of two Cauchy-like matrices is Cauchy-like under some circumstances.

Questions

- Given a family of matrices T, what is the optimal choice of the family (A, B)?
- If A and B are lower triangular Toeplitz matrices we have a fast Schur algorithm. What family of matrices does this correspond to?
- Is there a generic technique to compute Tx quickly?!
- Is there a generic technique to handle structured matrices of the form $T_1 \otimes T_2 + T_3 \otimes T_4$?

Low rank structure

- Simple rank structures
- Sparse matrices
- Sparsification

Simple rank structures

The Schur algorithm exploits low-rank structure in an oblique manner. Many modern algorithms are built on a few trivial observations:

- If $A = PQ^T$ with $P, Q \in \mathbb{R}^{n \times r}$, then $Ax = P(Q^T x)$ can be computed in $O(nr)$ operations.
- Furthermore P and Q can be computed in $O(n^2 r)$ operations via Gaussian elimination.
- $(I + PQ^T)^{-1}x = (I - P(I + Q^T P)^{-1}Q^T)x$ which can be computed in $O(nr^2)$ operations.
- A nice coherent presentation is possible via sparse matrices.

Sparse matrices

- What is surprising is that these algorithms are closely tied to sparse Gaussian elimination.
- For example Gaussian elimination on a matrix of bandwidth k costs only $O(nk^2)$ operations.
- Ordering is critical for fast sparse Gaussian elimination. For the arrowhead matrix

$$\begin{bmatrix} \clubsuit & 0 & \cdots & 0 & \clubsuit \\ 0 & \clubsuit & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \clubsuit \\ 0 & \cdots & 0 & \clubsuit & \clubsuit \\ \clubsuit & \cdots & \clubsuit & \clubsuit & \clubsuit \end{bmatrix}$$

  sparse Gaussian elimination takes linear time. However, if we reverse the ordering of both rows and columns, sparse Gaussian elimination take cubic time!

Sparsification (1/2)

Let's **discover** the formula for $(D + PQ^T)^{-1}$, with $P, Q \in \mathbb{R}^{n \times r}$. First we see how to quickly multiply $b = (D + PQ^T)x$

$$y = Q^T x \qquad b = Dx + Py$$

Our goal is to find x given b. Note that the above formulas are linear and can be expressed in matrix form

$$\begin{bmatrix} D & P \\ -Q^T & I \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

where all unknown quantities are on the left. If we eliminate y first we see that $D + PQ^T$ is the dense Schur complement of a sparse (identity sub-block) matrix. However, we can eliminate in the **opposite** order instead.

Sparsification (2/2)

$$\begin{bmatrix} D & P \\ 0 & I + Q^T D^{-1} P \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ Q^T D^{-1} b \end{bmatrix}$$

Now fast back-substituion yields

$$y = (I + Q^T D^{-1} P)^{-1} Q^T D^{-1} b \qquad x = D^{-1}(b - Py)$$

which is very efficient if D is a structured matrix.
Plugging y back in we discover the well-known formula

$$x = (D^{-1} - D^{-1} P (I + Q^T D^{-1} P)^{-1} Q^T D^{-1}) b$$

This is the core of the strategy we will follow.

Sequentially Semi-Separable Representations

- Hankel blocks
- Representation
- Controllability & Observability
- Banded matrices
- Matrix–vector multiply
- Diagonal representation
- Inversion
- $\pm$
- Multiplication
- Lower times lower
- Lower times upper
- Inverse of lower
- LU factorization
- Inverse

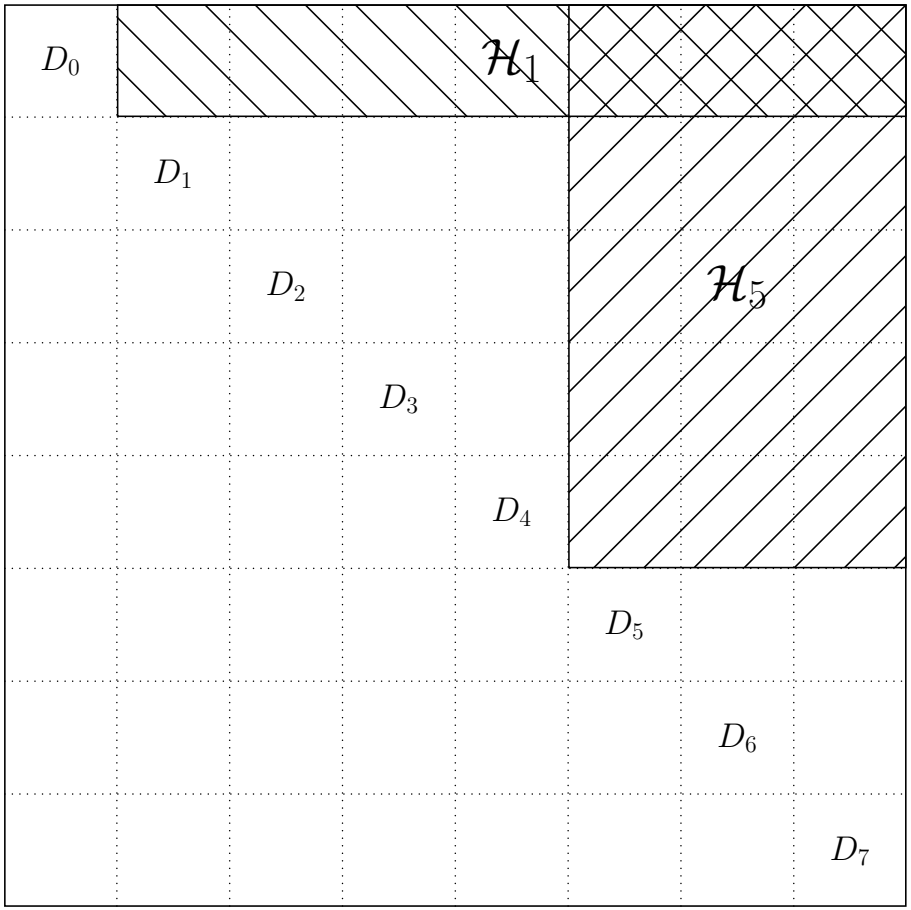
Hankel blocks (1/2)

For a square matrix A consider the block partition

$$A = \begin{array}{c} \\ m \\ n-m \end{array} \begin{array}{c} \begin{array}{cc} m & n-m \end{array} \\ \begin{pmatrix} \clubsuit & A_{01} \\ A_{10} & \clubsuit \end{pmatrix} \end{array}$$

- We will refer to the off-diagonal blocks, $A_{01} = \mathcal{H}_m$ and $A_{10} = \mathcal{G}_m$, as Hankel blocks.
- Hankel blocks never include entries on the principal diagonal.
- Now suppose that we have a family of matrices A for which **every** Hankel block has rank at most r independent of the size n. Can we do matrix arithmetic quickly with this family?
- Answer: Yes, via SSS representations [1, 5].
- Call r the **SSS-rank** of A.

Hankel blocks (2/2)



Representation

If A has SSS-rank at most r then there exists sequences of matrices D_i, P_i, Q_i, R_i, U_i, V_i and W_i, such that each is of size no more than r, and

$$A = \begin{bmatrix} D_0 & U_0V_1^T & U_0W_1V_2^T & U_0W_1W_2V_3^T & \cdots \\ P_1Q_0^T & D_1 & U_1V_2^T & U_1W_2V_3^T & \cdots \\ P_2R_1Q_0^T & P_2Q_1^T & D_2 & U_2V_3^T & \cdots \\ P_3R_2R_1Q_0^T & P_3R_2Q_1^T & P_3Q_2^T & D_3 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

The converse is also true.

Controllability & Observability

- For the converse direction note that

$$\mathcal{H}_j = \begin{bmatrix} U_0 W_1 W_2 \cdots W_{j-1} \\ U_1 W_2 \cdots W_{j-1} \\ \vdots \\ U_{j-2} W_{j-1} \\ U_{j-1} \end{bmatrix} \begin{bmatrix} V_j^T & W_j V_{j+1}^T & W_j W_{j+1} V_{j+2}^T & \cdots \end{bmatrix}$$

- For the forward direction note that compatible low-rank factorizations of $\mathcal{H}_j$ defines U_{j-1}, W_{j-1} and V_j recursively in terms of those from $\mathcal{H}_{j-1}$.
- This yields an $O(n^2 r)$ algorithm for constructing the SSS representation of a general dense matrix.
- Faster algorithms are available in special cases.

Banded matrices

A **tri-diagonal** matrix T, has SSS-rank at most 1. An SSS representation can be written down explicitly.

$$\begin{aligned} D_i &= T_{i,i} \\ U_i &= T_{i,i+1} \\ W_i &= 0 \\ V_i &= 1 \\ P_i &= T_{i,i-1} \\ R_i &= 0 \\ Q_i &= 1 \end{aligned}$$

The inverse of a tri-diagonal matrix is **not** tri-diagonal but it has SSS-rank at most 1 too as we will see later.

Matrix–vector multiply

To compute

$$\begin{bmatrix} D_0 & U_0V_1^T & U_0W_1V_2^T & \cdots \\ P_1Q_0^T & D_1 & U_1V_2^T & \cdots \\ P_2R_1Q_0^T & P_2Q_1^T & D_2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \end{bmatrix}$$

we can proceed as follows:

$$\begin{aligned} g_n &= V_n^T x_n \\ g_j &= V_j^T x_j + W_j g_{j+1}, \qquad j < n, \\ h_0 &= Q_0^T x_0 \\ h_j &= R_j h_{j-1} + Q_j^T x_j, \qquad 0 < j, \\ b_i &= P_i h_{i-1} + D_i x_i + U_i g_{i+1}. \end{aligned}$$

Diagonal representation (1/3)

Let

$$W = \begin{bmatrix} W_0 & & \\ & W_1 & \\ & & \ddots \end{bmatrix} = \mathrm{diag}\{W_i\} \qquad g = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \end{bmatrix}$$

and $V = \mathrm{diag}\{V_i\}$. Then the recursions for g can be written as

$$(I - W Z_{\downarrow}^T) g = V^T x$$

Similarly the recursions for h can be written as

$$(I - R Z_{\downarrow}) h = Q^T x$$

and those for b as

$$b = U Z_{\downarrow}^T g + D x + P Z_{\downarrow} h$$

Diagonal representation (2/3)

We can assemble all the recursions into a single matrix equation:

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

Eliminating g and h we get the diagonal representation of A as a Schur complement of the above sparse matrix

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

- Since $WZ_{\downarrow}^T$ and $RZ_{\downarrow}$ are bi-diagonal matrices this also makes obvious the fast matrix–vector multiplication algorithm.
- Also note that the formula splits into strictly lower-triangular, diagonal and strictly upper-triangular terms.

Diagonal representation (3/3)

We see the key algebraic identity is the formula

$$\begin{bmatrix} I & -W_0 & 0 & \cdots \\ 0 & I & -W_1 & \ddots \\ \vdots & \ddots & \ddots & \ddots \end{bmatrix}^{-1} = \begin{bmatrix} I & W_0 & W_0W_1 & W_0W_1W_2 & \cdots \\ 0 & I & W_1 & W_1W_2 & \cdots \\ \vdots & \ddots & \ddots & \ddots & \end{bmatrix}$$

which is also another way to express the fast multiplication.

Inversion (1/2)

The equations

$$\begin{bmatrix} D & UZ_{\downarrow}^T & PZ_{\downarrow} \\ -V^T & I - W\overset{\downarrow}{Z}_{\downarrow}^T & 0 \\ -Q^T & 0 & I - RZ_{\downarrow} \end{bmatrix} \begin{bmatrix} x \\ g \\ h \end{bmatrix} = \begin{bmatrix} b \\ 0 \\ 0 \end{bmatrix}$$

can be re-ordered to yield almost no fill-in during sparse Gaussian elimination. First re-order the unknowns

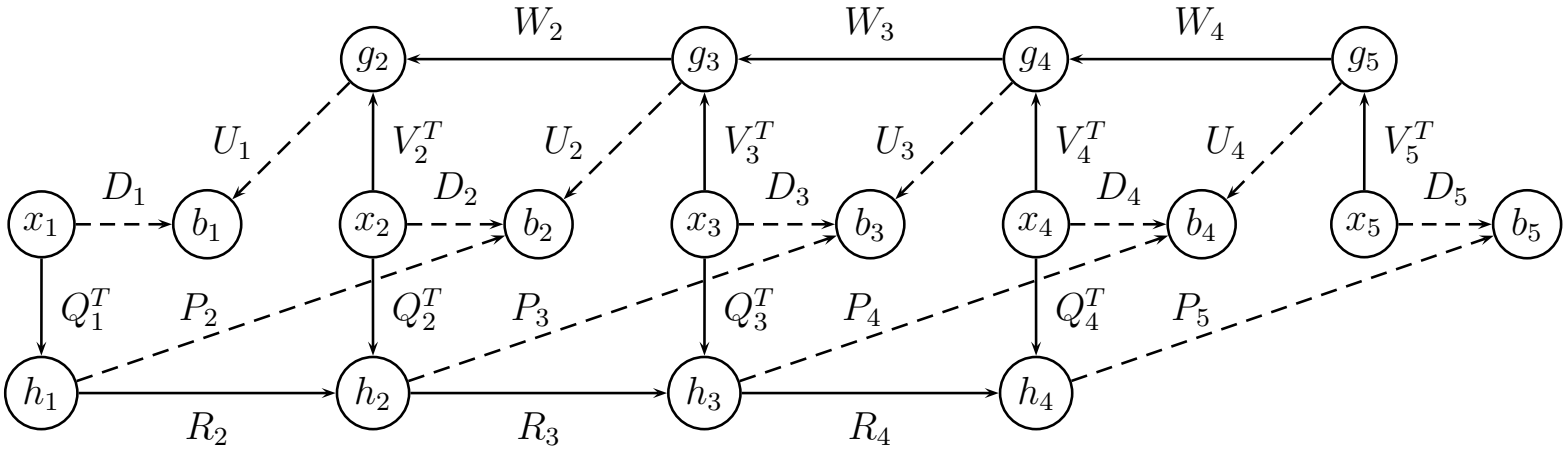
$$y_i = \begin{bmatrix} x_i \\ g_i \\ h_i \end{bmatrix}$$

Then observe that the equations can be re-ordered as

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

Inversion (2/2)

- This is a block tri-diagonal matrix.
- So there exists a numerically stable $O(nr^2)$ algorithm to compute x from b.
- The whole affair is more transparent by looking at the associated data-flow graph:



± (1/3)

- From
$$\begin{bmatrix} \clubsuit & A_{01} \\ \clubsuit & \clubsuit \end{bmatrix} \pm \begin{bmatrix} \clubsuit & B_{01} \\ \clubsuit & \clubsuit \end{bmatrix} = \begin{bmatrix} \clubsuit & A_{01} \pm B_{01} \\ \clubsuit & \clubsuit \end{bmatrix}$$
we see that the SSS-rank of $A \pm B$ is at most the sum of the SSS-ranks of A and B.
- The SSS representation of $A \pm B$ can be computed in linear time from that of A and B.
- The algorithm is easily worked out by brute force, but we introduce an algebraic technique [Dewilde & van der Veen].
- It is enough to consider the problem for two strictly lower triangular matrices in diagonal form:
$$P_1 Z_\downarrow (I - R_1 Z_\downarrow)^{-1} Q_1^T \pm P_2 Z_\downarrow (I - R_2 Z_\downarrow)^{-1} Q_2^T$$

- Note that the previous equation can be re-written as:

$$\begin{bmatrix} P_1 Z_\downarrow & P_2 Z_\downarrow \end{bmatrix} \mathcal{S}^T \mathcal{S} \begin{bmatrix} I - R_1 Z_\downarrow & 0 \\ 0 & I - R_2 Z_\downarrow \end{bmatrix}^{-1} \mathcal{S}^T \mathcal{S} \begin{bmatrix} Q_1 & \pm Q_2 \end{bmatrix}^T$$

  where $\mathcal{S}$ is the **perfect shuffle** permutation matrix ($\mathcal{S}^T \mathcal{S} = I$).
- With some abuse of notation we will continue to use $Z_\downarrow$ to denote a "block" down-shift matrix where the inner partitions will be chosen by the context!

$$\mathcal{S} \begin{bmatrix} Z_\downarrow & 0 \\ 0 & Z_\downarrow \end{bmatrix} \mathcal{S}^T = Z_\downarrow$$

- Let

$$\begin{bmatrix} P_1 & P_2 \end{bmatrix} \mathcal{S}^T = P_3 \qquad \mathcal{S} \begin{bmatrix} Q_1 & \pm Q_2 \end{bmatrix}^T = Q_3^T \qquad \mathcal{S}^T \begin{bmatrix} R_1 & 0 \\ 0 & R_2 \end{bmatrix} \mathcal{S} = R_3$$

Now the representation of AB simplifies to

$$AB = P_3 Z_{\downarrow} (I - R_3 Z_{\downarrow})^{-1} Q_3^T$$

and observe that everything is still diagonal which yields the fast algorithm

$$\begin{aligned} P_{3;i} &= \begin{bmatrix} P_{1;i} & P_{2;i} \end{bmatrix} \\ Q_{3;i} &= \begin{bmatrix} Q_{1;i} & Q_{2;i} \end{bmatrix} \\ R_{3;i} &= \begin{bmatrix} R_{1;i} & 0 \\ 0 & R_{2;i} \end{bmatrix} \end{aligned}$$

This trivial algorithm could have also been obtained by a "state-space" argument using the data-flow graph.

Multiplication

Note that

$$\begin{bmatrix} \clubsuit & A_{01} \\ \clubsuit & \clubsuit \end{bmatrix} \begin{bmatrix} \clubsuit & B_{01} \\ \clubsuit & \clubsuit \end{bmatrix} = \begin{bmatrix} \clubsuit & A_{01}\clubsuit + \clubsuit B_{01} \\ \clubsuit & \clubsuit \end{bmatrix}$$

- The SSS-rank of AB is at most the sum of the SSS-ranks of A and B.
- There exists an $O(n)$ operations recursion to compute the SSS representation of AB from that of A and B.
- We use the algebraic approach and split the problem into four pieces of which only two will be covered in these slides.
  - Strictly **lower** triangular times strictly **lower** triangular
  - Strictly **lower** triangular times strictly **upper** triangular

Lower times lower

We multiply out the diagonal representation for strictly lower:

$$\begin{aligned}
P_1 Z_\downarrow (I - R_1 Z_\downarrow)^{-1} Q_1^T P_2 Z_\downarrow (I - R_2 Z_\downarrow)^{-1} Q_2^T &= \\
\begin{bmatrix} 0 & P_1 Z_\downarrow \end{bmatrix} \begin{bmatrix} I - R_2 Z_\downarrow & 0 \\ -Q_1^T P_2 Z_\downarrow & I - R_1 Z_\downarrow \end{bmatrix}^{-1} \begin{bmatrix} Q_2^T \\ 0 \end{bmatrix} &= \\
\begin{bmatrix} 0 & P_1 Z_\downarrow \end{bmatrix} \mathcal{S}^T \mathcal{S} \left(I - \begin{bmatrix} R_2 & 0 \\ Q_1^T P_2 & R_1 \end{bmatrix} \begin{bmatrix} Z_\downarrow & 0 \\ 0 & Z_\downarrow \end{bmatrix} \right)^{-1} \mathcal{S}^T \mathcal{S} \begin{bmatrix} Q_2^T \\ 0 \end{bmatrix} &= \\
P_3 Z_\downarrow (I - R_3 Z_\downarrow)^{-1} Q_3^T &
\end{aligned}$$

where

$$P_{3;i} = \begin{bmatrix} 0 & P_{1;i} \end{bmatrix} \qquad Q_{3;i} = \begin{bmatrix} Q_{2;i} & 0 \end{bmatrix} \qquad R_{3;i} = \begin{bmatrix} R_{2;i} & 0 \\ Q_{1;i} P_{2;i} & R_{1;i} \end{bmatrix}$$

This fast algorithm is harder to infer by other means.

Lower times upper (1/4)

Mutiplying out the diagonal representation:

$$PZ_{\downarrow}(I - RZ_{\downarrow})^{-1}Q^{T}UZ_{\downarrow}^{T}(I - WZ_{\downarrow}^{T})^{-1}V^{T}$$

The previous technique does not work now, so we look for a PFE similar to:

$$\frac{z^{-1}}{(1-\alpha z)(1-\beta z^{-1})} = \frac{\alpha(1-\alpha\beta)^{-1}}{1-\alpha z} + \frac{(1-\alpha\beta)^{-1}z^{-1}}{1-\beta z^{-1}}$$

We try

$$\begin{aligned}(I - RZ_{\downarrow})^{-1}Q^{T}UZ_{\downarrow}^{T}(I - WZ_{\downarrow}^{T})^{-1} \quad &= \\ (I - RZ_{\downarrow})^{-1}A + BZ_{\downarrow}^{T}(I - WZ_{\downarrow}^{T})^{-1} &\end{aligned}$$

with A and B as block diagonal matrices.

Lower times upper (2/4)

Some algebra reveals that

$$A = RZ_{\downarrow}BZ_{\downarrow}^{T} \qquad Q^{T}U = RZ_{\downarrow}BZ_{\downarrow}^{T}W - B$$

Exploiting the diagonal nature we obtain B via the recursion:

$$B_1 = Q_1^{T}U_1 \qquad B_i = Q_i^{T}U_i + R_{i-1}B_{i-1}W_i$$

From this we can recover A and hence:

$$PZ_{\downarrow}(I - RZ_{\downarrow})^{-1}Q^{T}UZ_{\downarrow}^{T}(I - WZ_{\downarrow}^{T})^{-1}V^{T} =$$
$$PZ_{\downarrow}(I - RZ_{\downarrow})^{-1}AV^{T} + PZ_{\downarrow}BZ_{\downarrow}^{T}(I - WZ_{\downarrow}^{T})^{-1}V^{T}$$

The second term is not in the desired form.

Lower times upper (3/4)

Using the trivial identity

$$(I - WZ_{\downarrow}^T)^{-1} = I + WZ_{\downarrow}^T(I - WZ_{\downarrow}^T)^{-1}$$

we have

$$PZ_{\downarrow}BZ_{\downarrow}^T(I - WZ_{\downarrow}^T)^{-1}V^T = PZ_{\downarrow}BZ_{\downarrow}^TV^T + (PZ_{\downarrow}BZ_{\downarrow}^TW)Z_{\downarrow}^T(I - WZ_{\downarrow}^T)^{-1}V^T$$

- Observe that $Z_{\downarrow}BZ_{\downarrow}^T$ is a block diagonal matrix.
- So we are in the desired diagonal form now and we can read off the algorithm for lower times upper.

Lower times upper (4/4)

We use a suffix 3 to denote the SSS representation of the product of a lower and upper SSS matrix.

$$
\begin{aligned}
D_3 &= PZ_{\downarrow}BZ_{\downarrow}^T V^T \\
P_3 &= P \\
Q_3 &= Q \\
R_3 &= R \\
U_3 &= PZ_{\downarrow}BZ_{\downarrow}^T W \\
V_3 &= V \\
W_3 &= W
\end{aligned}
$$

- This requires $O(n)$ operations.
- The SSS-rank does **not** increase.

Inverse of lower (1/2)

Claim: The SSS-rank of a lower triangular matrix is at most that of the original matrix.
Follows from:

$$\begin{bmatrix} \clubsuit & 0 \\ A_{10} & \clubsuit \end{bmatrix}^{-1} = \begin{bmatrix} \clubsuit & 0 \\ \clubsuit A_{10} \clubsuit & \clubsuit \end{bmatrix}$$

To find the representation quickly consider the diagonal form:

$$(D + PZ_{\downarrow}(I - RZ_{\downarrow})^{-1}Q^T)^{-1} =$$
$$D^{-1} - D^{-1}PZ_{\downarrow}(I - (R - Q^T D^{-1}P)Z_{\downarrow})^{-1}Q^T D^{-1}$$

Inverse of lower (2/2)

We can now read off the algorithm, using subscript 2 to denote quantities belonging to the inverse.

$$\begin{aligned} D_2 &= D^{-1} \\ P_2 &= -D^{-1}P \\ Q_2 &= D^{-T}Q \\ R_2 &= R - Q^T D^{-1} P \end{aligned}$$

This requires $O(n)$ operations.

LU factorization (1/4)

Claim: The SSS-ranks of the LU factors are at most that of the original matrix.
Follows from:

$$\begin{bmatrix} \clubsuit & A_{01} \\ A_{10} & \clubsuit \end{bmatrix} = \begin{bmatrix} \clubsuit & 0 \\ A_{10}\clubsuit & \clubsuit \end{bmatrix} \begin{bmatrix} \clubsuit & \clubsuit A_{01} \\ 0 & \clubsuit \end{bmatrix}$$

- To find the algorithm we can try to *solve* the recursions for the fast multiplication of lower times upper triangular matrices in SSS form.
- An alternative is a nice approach proposed by Ming Gu (UCB)

LU factorization (2/4)

First we generalize the representation a little bit by including an extra term:

$$B = A + \begin{bmatrix} P_0 \\ P_1 R_0 \\ P_2 R_1 R_0 \\ \vdots \end{bmatrix} C_0 \begin{bmatrix} V_0 \\ V_1 W_0^T \\ V_2 W_1^T W_0^T \\ \vdots \end{bmatrix}^T$$

with $C_0 = 0$ (but pretend it is not). Note that we can recover the first row and column of B:

$$\begin{bmatrix} D_0 + P_0 C_0 V_0^T & (U_0 + P_0 C_0 W_0) V_1^T & (U_0 + P_0 C_0 W_0) W_1 V_2^T & \cdots \\ P_1 (Q_0^T + R_0 C_0 V_0^T) & \cdot & \cdot & \\ P_2 R_1 (Q_0^T + R_0 C_0 V_0^T) & \cdot & \cdot & \\ \vdots & & & \end{bmatrix}$$

This gives the first column of L and the first row of U.

Next the Schur complement.

$$\begin{bmatrix} D_1 & U_1V_2^T & U_1W_2V_3^T & \cdots \\ P_2Q_1^T & D_2 & U_2V_3^T & \cdots \\ P_3R_2Q_1^T & P_3Q_2^T & D_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$+ \begin{bmatrix} P_1 \\ P_2R_1 \\ P_3R_2R_1 \\ \vdots \end{bmatrix} R_0C_0W_0 \begin{bmatrix} V_1^T & W_1V_2^T & W_1W_2V_3^T & \cdots \end{bmatrix}$$

$$- \begin{bmatrix} P_1 \\ P_2R_1 \\ P_3R_2R_1 \\ \vdots \end{bmatrix} (Q_0^T + R_0C_0V_0^T)(D_0 + P_0C_0V_0^T)^{-1}(U_0 + P_0C_0W_0) \begin{bmatrix} V_1 \\ V_2W_1^T \\ V_3W_2^TW_1^T \\ \vdots \end{bmatrix}^T$$

LU factorization (4/4)

Clearly the only thing we need to do is

$$C_1 = R_0 C_0 W_0 - (Q_0^T + R_0 C_0 V_0^T)(D_0 + P_0 C_0 V_0^T)^{-1}(U_0 + P_0 C_0 W_0)$$

Doing this recursively we can compute the LU factorization in $O(n)$ operations.
The SSS structure of L and U can be inferred from the above formulas.

Inverse

- From

$$\begin{bmatrix} \clubsuit & A_{01} \\ A_{10} & \clubsuit \end{bmatrix}^{-1} = \begin{bmatrix} \clubsuit & \clubsuit A_{01} \clubsuit \\ \clubsuit A_{10} \clubsuit & \clubsuit \end{bmatrix}$$

  we see that the SSS rank of A^{-1} is at most the SSS rank of A.
- Using our previous formulas the SSS representation of A^{-1} can be computed in linear time from the SSS representation of the LU factors of A.

Hierarchically semi-separable representations

- Fast multi-pole method
- Row and column Hankel blocks
- Binary partition tree
- Row and column bases
- Expansion coefficients
- Representation
- Construction
- Matrix–vector multiply
- Multiply signal flow graph
- Solver
- Multiplication
- LU factorization
- Lower inverse
- Inverse

Fast multi-pole method

- The FMM [7] can evaluate sums of the following form

$$p_i = \sum_{j=0}^{n-1} \frac{q_j}{1 + (x_i - y_j)^2}$$

  in $O(n \log_2 n)$ operations when $x_i, y_i \in \mathbb{R}$.
- In matrix terms this is an algorithm for the fast matrix–vector multiplication of the matrix $A_{ij} = (1 + (x_i - y_j)^2)^{-1}$.
- The FMM can also be viewed as a matrix representation that captures certain low-rank structures in A.
- The main question is how to apply A^{-1} quickly?
- The answer is the HSS representation [2].

Row and column Hankel blocks (1/6)

The HSS representation is closely related to the SSS representation but uses slightly different Hankel blocks.

$$A = \begin{array}{c} \\ l \\ m \\ n-l-m \end{array} \begin{array}{c} \begin{array}{ccc} l & m & n-l-m \end{array} \\ \begin{pmatrix} \clubsuit & A_{01} & \clubsuit \\ A_{10} & \clubsuit & A_{12} \\ \clubsuit & A_{21} & \clubsuit \end{pmatrix} \end{array}$$

- We call

$$\mathcal{R}_l = \begin{bmatrix} A_{10} & A_{12} \end{bmatrix}$$

  as a **row** Hankel block.
- We call

$$\mathcal{C}_l = \begin{bmatrix} A_{01} \\ A_{21} \end{bmatrix}$$

  as a **column** Hankel block.

Row and column Hankel blocks (2/6)

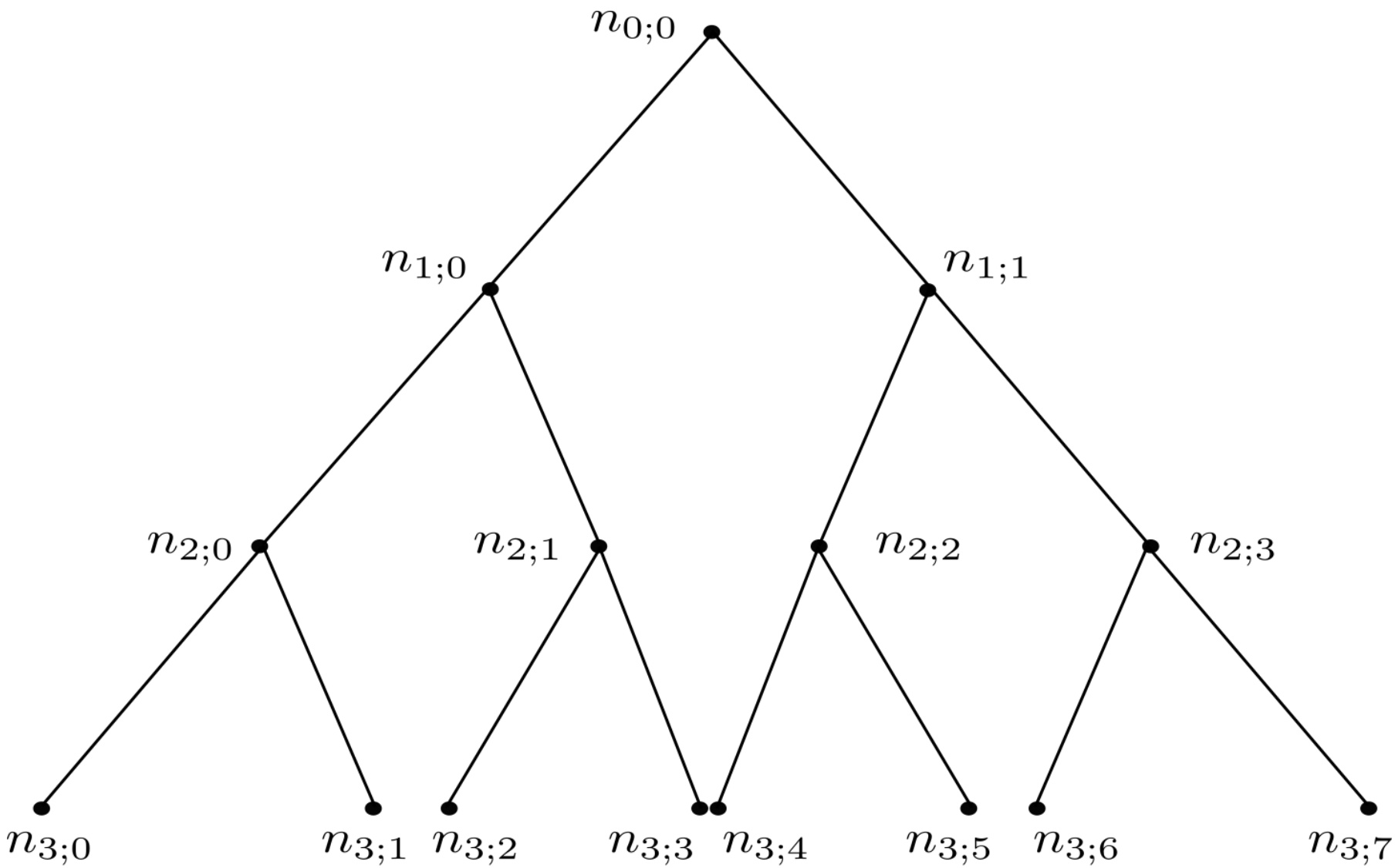
- The HSS representation captures the low ranks of both row and column Hankel blocks but **not** all of them. It only uses a pre-specified set of hierarchical partitions.
- Let $n_{0;0} = n$. We assume that there are non-negative integers $n_{k;i}$ specified such that

$$n_{k;i} = n_{k+1;2i} + n_{k+1;2i+1}, \qquad 0 \le k < K, \qquad 0 \le i < 2^k$$

  Usually these numbers are visualized on a binary tree with K levels that is called the partition tree.
- Using the non-negative integers $n_{k;i}$ on a single level k we can partition the matrix

$$A = A_k = \begin{array}{c} \\ n_{k;0} \\ n_{k;1} \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccc} n_{k;0} & n_{k;1} & \cdots \end{array} \\ \begin{pmatrix} A_{k;0,0} & A_{k;0,1} & \cdots \\ A_{k;1,0} & A_{k;1,1} & \cdots \\ \vdots & \vdots & \end{pmatrix} \end{array}$$

Binary partition tree



Row and column Hankel blocks (3/6)

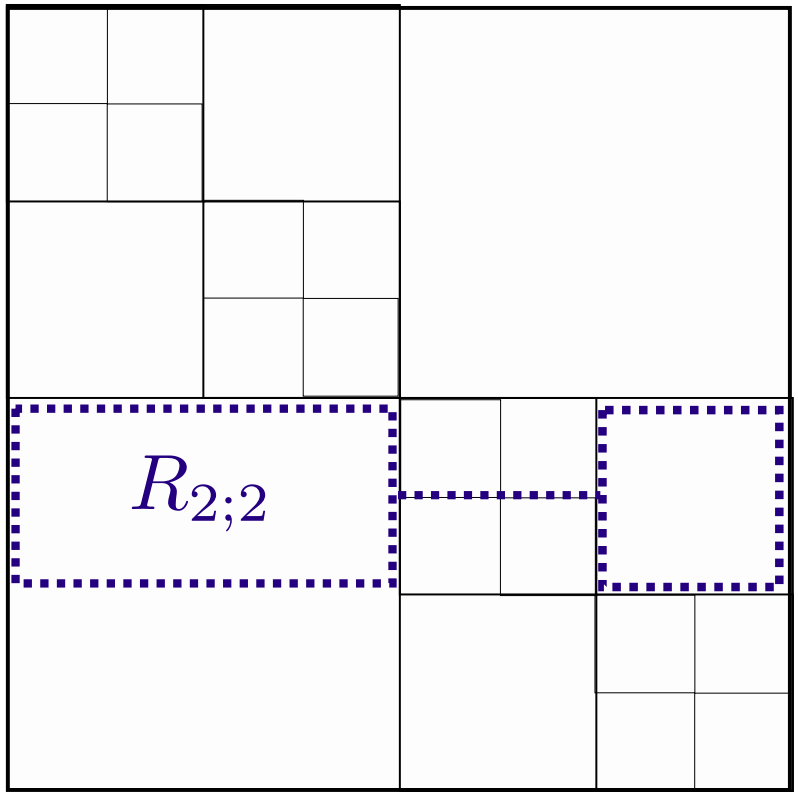
The corresponding row Hankel block

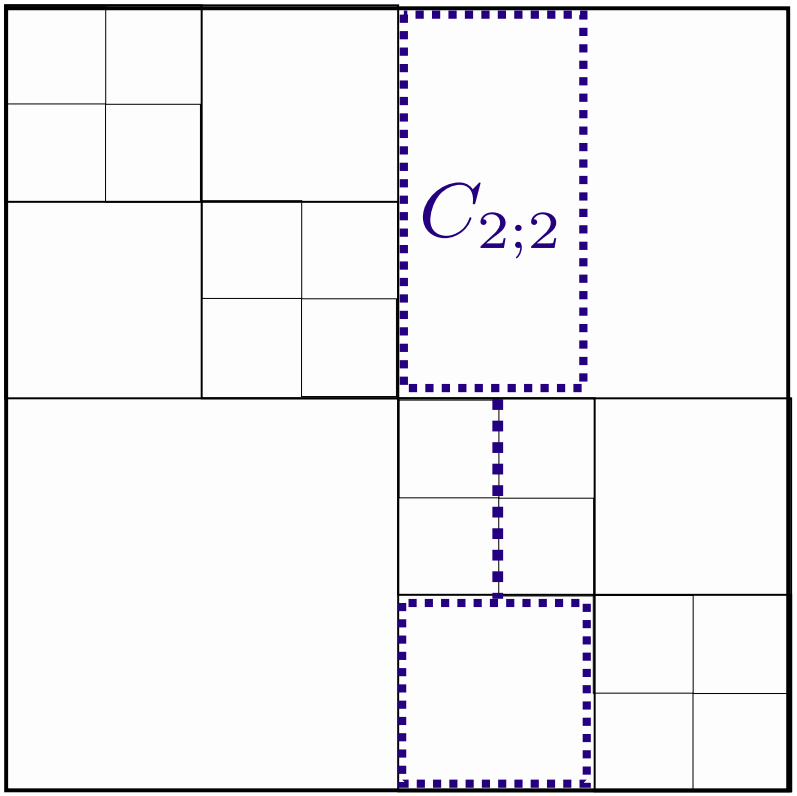
$$\mathcal{R}_{k;i} = \begin{bmatrix} A_{k;i,0} & \cdots & A_{k;i,i-1} & A_{k;i,i+1} & \cdots A_{k;i,2^k-1} \end{bmatrix}$$

and column Hankel block

$$\mathcal{C}_{k;i} = \begin{bmatrix} A_{k;0,i} \\ \vdots \\ A_{k;i-1,i} \\ A_{k;i+1,i} \\ \vdots \\ A_{k;2^k-1,i} \end{bmatrix}$$

Row and column bases (4/6)





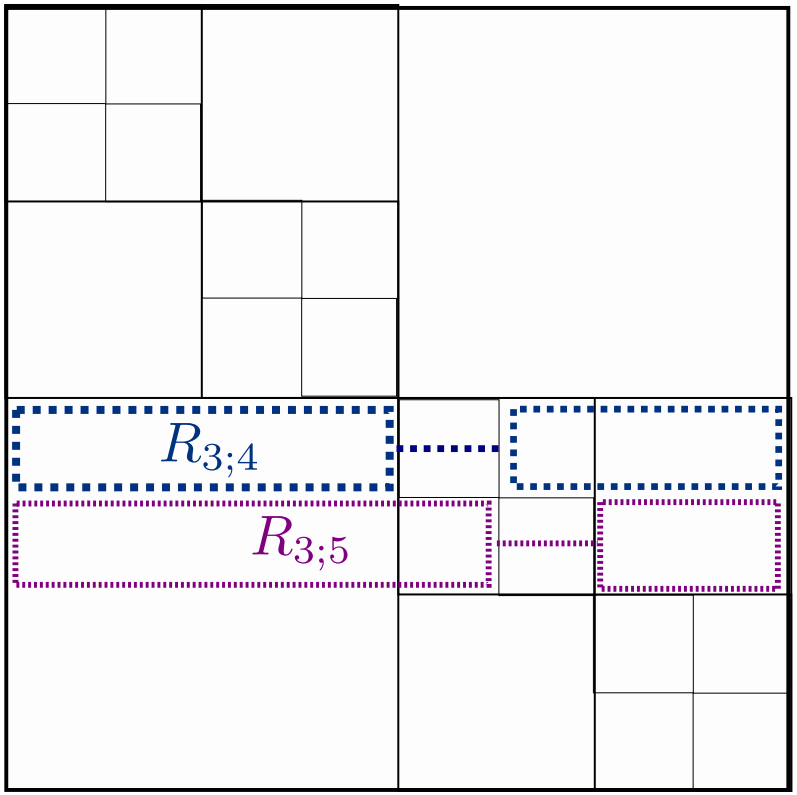
Row and column Hankel blocks (5/6)

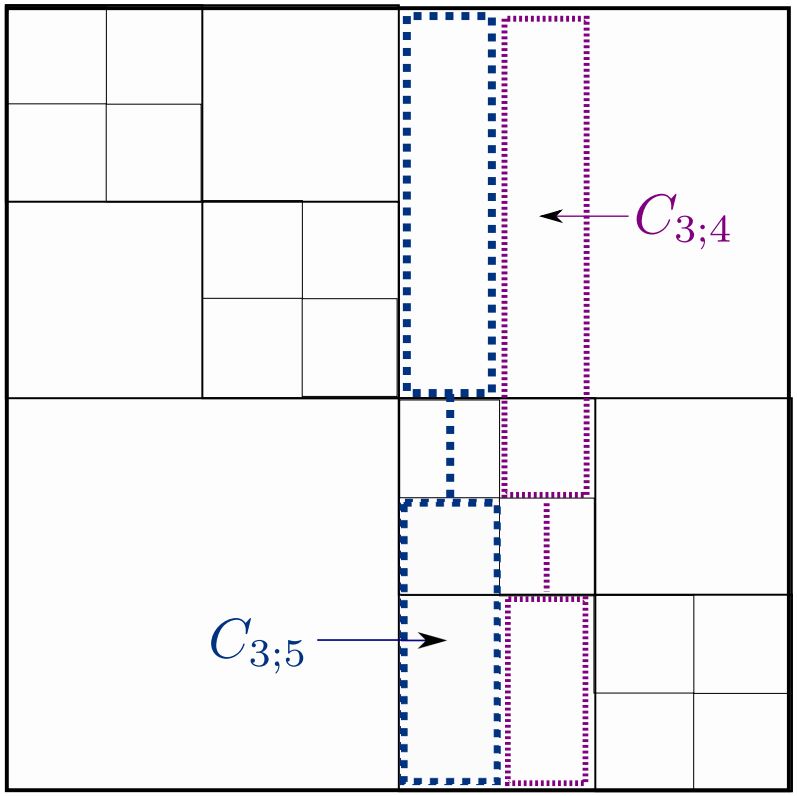
- Row Hankel blocks overlaps those of children node:

$$\mathcal{R}_{k+1;i} = \begin{bmatrix} A_{k;2i,0} & \cdots & A_{k;2i,2(i-1)} & A_{k;2i,2(i+1)} & \cdots & A_{k;2i,2^k-1} \\ A_{k;2i+1,0} & \cdots & A_{k;2i+1,2(i-1)} & A_{k;2i+1,2(i+1)} & \cdots & A_{k;2i+1,2^k-1} \end{bmatrix}$$

- Similarly for column Hankel blocks.
- The HSS (FMM) representation exploits the low rank structure of individual row and column Hankel blocks and the overlap.
- We will call the maximum of the ranks of the row and column Hankel blocks as the **HSS-rank** of the matrix.
- This is similar to SSS but a bit simpler.

Row and column bases (6/6)





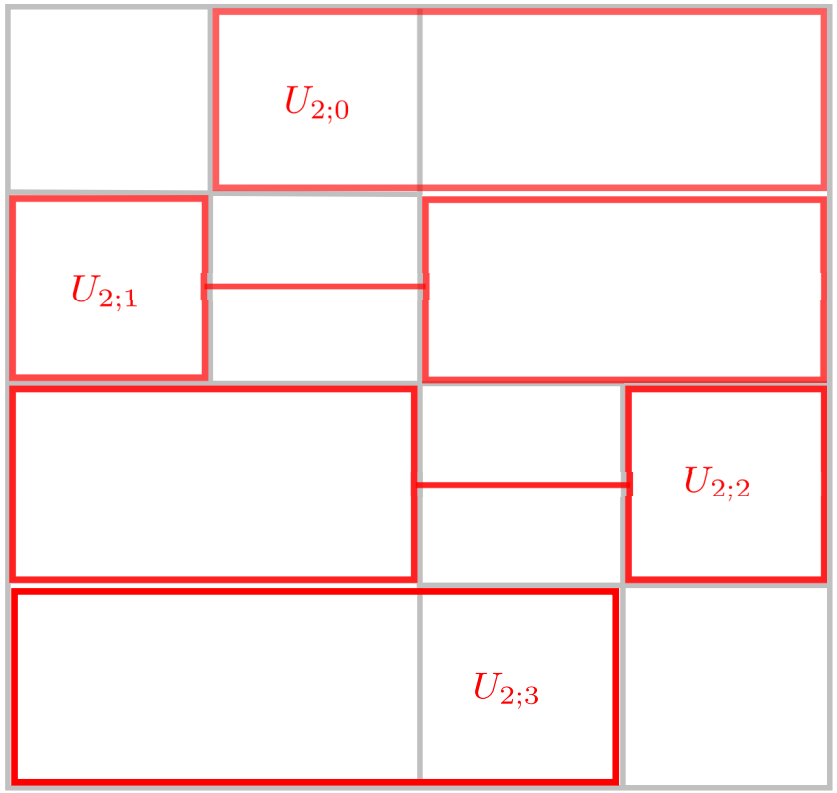
Row and column bases (1/3)

- Let $U_{k;i}$ be a full column rank matrix such that its range space is identical to that of $\mathcal{R}_{k;i}$.
- Let $V_{k;i}$ be a full column rank matrix such that its range space is identical to that of $\mathcal{C}_{k;i}^T$.
- Then there exists matrices $B_{k;i,j}$ such that

$$A_{k;i,j} = U_{k;i} B_{k;i,j} V_{k;i,j}^T, \qquad i \neq j$$

- This is a highly redundant representation and we can throw many of these matrices out.

Row and column bases (2/3)



Row and column bases (3/3)

- Due to the overlap of $\mathcal{R}_{k;i}$ with $\mathcal{R}_{k+1;2i}$ and $\mathcal{R}_{k+1;2i+1}$ it follows that there exists **translation** matrices $R_{k+1;2i}$ and $R_{k+1;2i+1}$ such that

$$U_{k;i} = \begin{bmatrix} U_{k+1;2i}R_{k+1;2i} \\ U_{k+1;2i+1}R_{k+1;2i+1} \end{bmatrix}$$

- Similarly for $\mathcal{C}_{k;i}$ we can define translation matrices $W_{k;i}$ such that

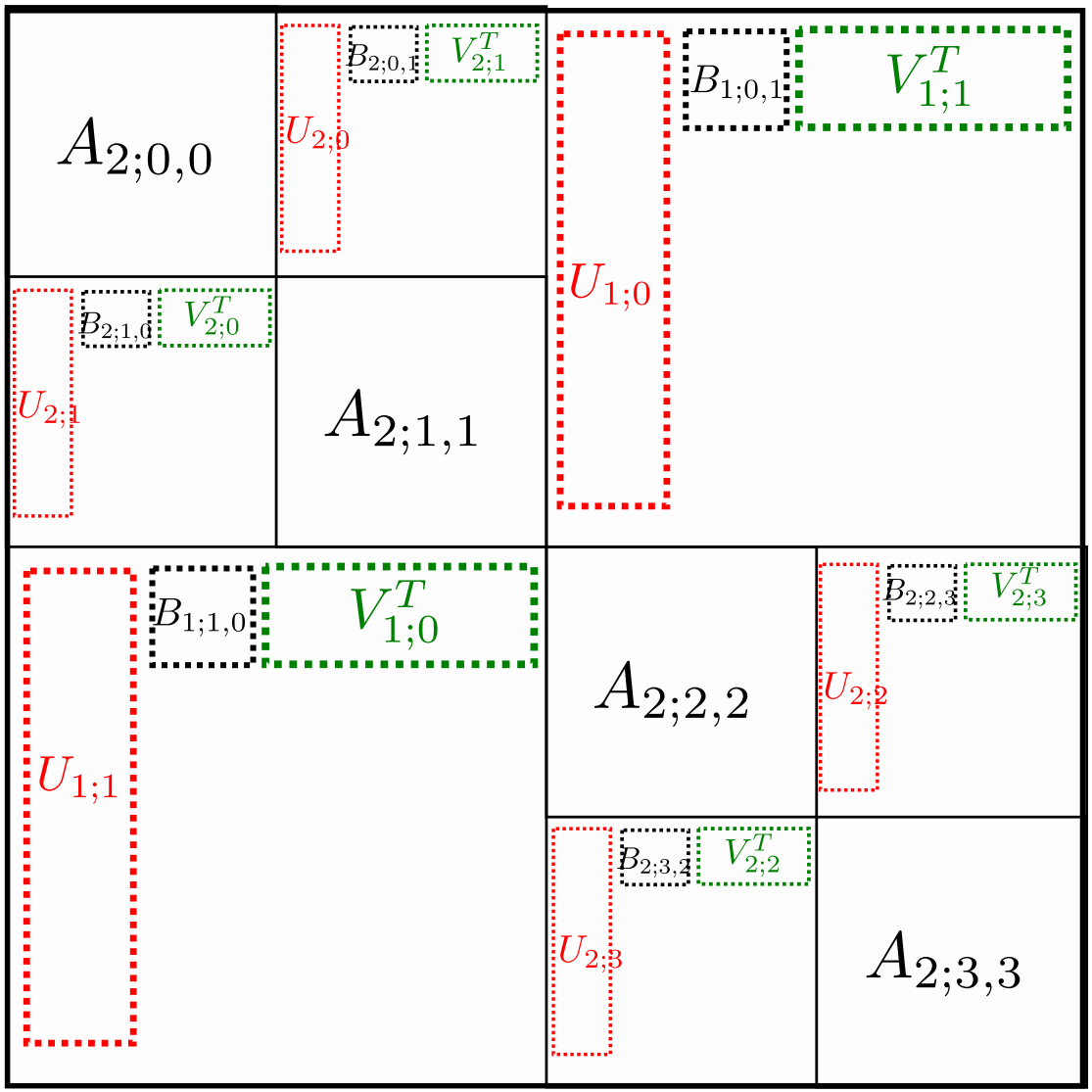
$$V_{k;i} = \begin{bmatrix} V_{k+1;2i}W_{k+1;2i} \\ V_{k+1;2i+1}W_{k+1;2i+1} \end{bmatrix}$$

Expansion coefficients (1/2)

- In the FMM representation one has to choose in addition which off-diagonal submatrices are represened explicitly in terms of $B_{k;i,j}$.
- For the HSS case a very simplistic choice is made. We only store $B_{k;2i,2i+1}$ and $B_{k;2i+1,2i}$.
- For example, note that $A_{1;0,1}$ covers all sub-matrices of the $A_{2;i,j}$ for $i = 0, 1$ and $j = 2, 3$. Therefore $B_{1;0,1}$ already *covers* those other blocks and their expansion coefficients are not needed.

$$A = \begin{bmatrix} \begin{bmatrix} A_{2;0,0} & U_{2;0}B_{2;0,1}V_{2;1}^T \\ U_{2;1}B_{2;1,0}V_{2;0}^T & A_{2;1,1} \end{bmatrix} & \begin{bmatrix} \\ U_{1;0}B_{1;0,1}V_{1;1}^T \\ \\ \end{bmatrix} \\ \begin{bmatrix} \\ U_{1;1}B_{1;1,0}V_{1;0}^T \\ \\ \end{bmatrix} & \begin{bmatrix} A_{2;2,2} & U_{2;2}B_{2;2,3}V_{2;3}^T \\ U_{2;3}B_{2;3,2}V_{2;2}^T & A_{2;3,3} \end{bmatrix} \end{bmatrix}$$

Expansion coefficients (2/2)

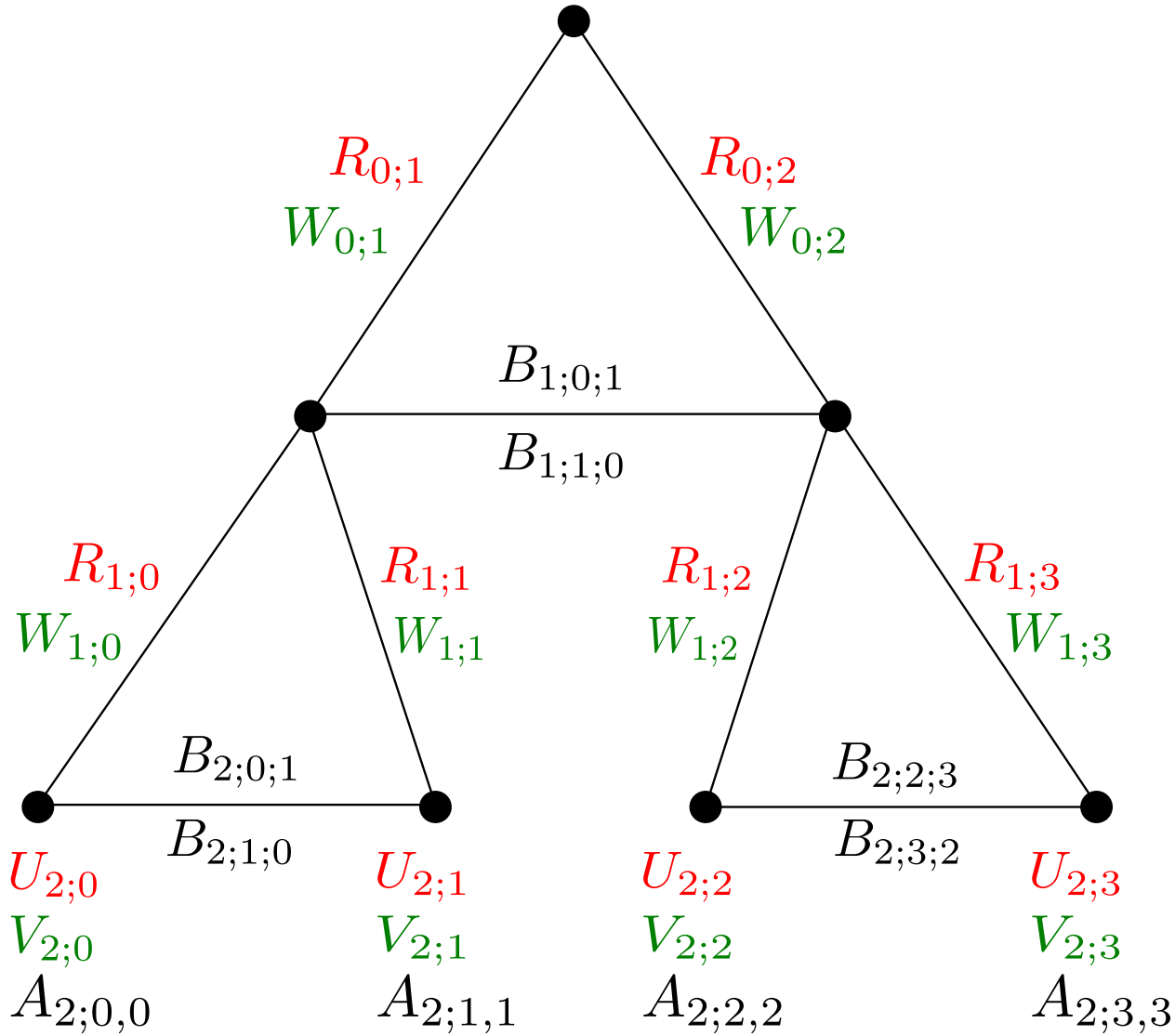


Representation (1/2)

We can now state the final HSS representation using the binary partition tree.

- On leaf nodes we *store* $A_{k;i,i}$, $U_{k;i}$, $V_{k;i}$.
- On the edge from parent (k, i) to child $(k+1, 2i)$, we store the translation operators $R_{k+1;2i}$, $W_{k+1;2i+1}$. Similarly for child $(k+1, 2i+1)$.
- On the edge from sibling $(k, 2i)$ to sibling $(k, 2i+1)$ we store $B_{k;2i,2i+1}$. Similarly for the opposite edge.

Representation (2/2)



Construction

- There is a generic construction algorithm, very similar to the SSS case, that takes $O(n^2)$ operations. It is best to do this algorithm over several sweeps.
- For special cases the representation can be computed much faster.
- For sparse matrices the HSS representation can be computed in linear time.
- If $A_{i,j} = f(x_i - y_j)$ then FMM techniques can be used to compute the representation in almost linear time.

Matrix–vector multiply (1/4)

Consider $Ax = b$. We partition x and b according to the HSS partition tree:

$$x_{0;0} = x, \qquad x_{k;i} = \begin{matrix} n_{k+1;2i} \\ n_{k+1;2i+1} \end{matrix} \begin{pmatrix} x_{k+1;2i} \\ x_{k+1;2i+1} \end{pmatrix}$$

Note that we have to compute $V_{k;i}^T x_{k;i} = g_{k;i}$. Clearly this is doable at leaf nodes. At non-leaf nodes we use the corresponding translation operators

$$\begin{aligned} g_{k;i} &= V_{k;i}^T x_{k;i} \\ &= \begin{bmatrix} W_{k+1;2i}^T V_{k+1;2i}^T & W_{k+1;2i+1}^T V_{k+1;2i+1}^T \end{bmatrix} \begin{bmatrix} x_{k+1;2i} \\ x_{k+1;2i+1} \end{bmatrix} \\ &= W_{k+1;2i}^T g_{k+1;2i} + W_{k+1;2i+1}^T g_{k+1;2i+1} \end{aligned}$$

Matrix–vector multiply (2/4)

Next we need to compute terms of the form $U_{k;2i}B_{k;2i,2i+1}g_{k;2i+1}$. The main idea is to delay these multiplications to save operations. To that end we define $f_{k;i}$ via

$$b_{k;i} = A_{k;i,i}\, x_{k;i} + U_{k;i} f_{k;i}$$

At leaf nodes we have access to $U_{k;i}$. At non-leaf nodes we use the corresponding translation operators instead. It is easier to develop the recursions by starting at the root level

$$b_{0;0} = A_{0;0,0}x_{0;0} + U_{0;0}f_{0;0} = Ax$$

which forces $f_{0;0} = [\,]$.

Matrix–vector multiply (3/4)

At level 1:

$$\begin{aligned} b_{1;0} &= A_{1;0,0}x_{1;0} + U_{1;0}B_{1;0,1}g_{1;1} + U_{1;0}R_{1;0}f_{0;0} \\ &= A_{1;0,0}x_{1;0} + U_{1;0}(B_{1;0,1}g_{1;1} + R_{1;0}f_{0;0}) \end{aligned}$$

which suggests that

$$f_{1;0} = B_{1;0,1}g_{1;1} + R_{1;0}f_{0;0}$$

The right recursions are

$$\begin{aligned} f_{k+1;2i} &= B_{k;2i,2i+1}\, g_{k;2i+1} + R_{k;2i}f_{k;i} \\ f_{k+1;2i+1} &= B_{k;2i+1,2i}\, g_{k;2i} + R_{k;2i+1}f_{k;i} \end{aligned}$$

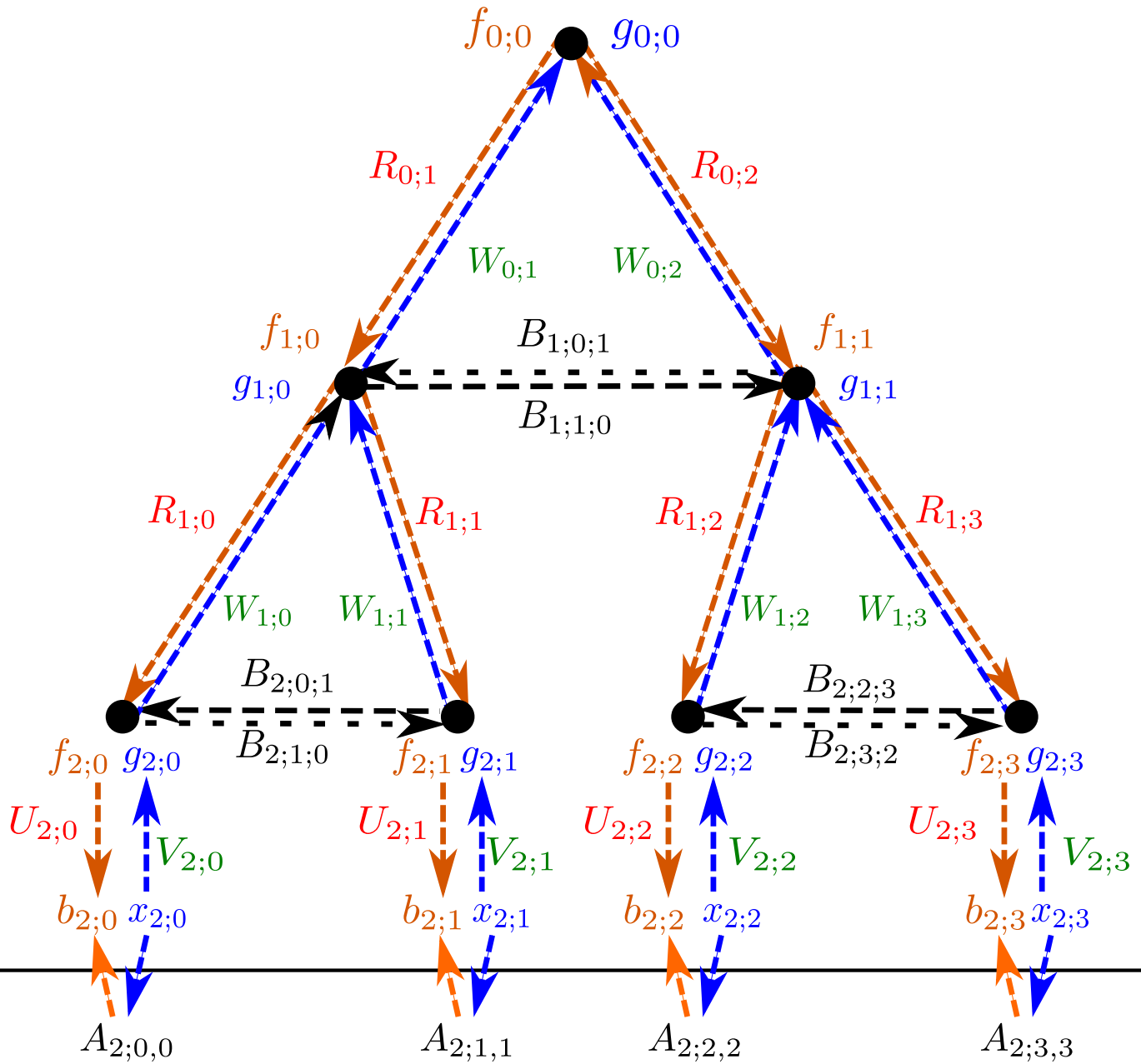
Matrix–vector multiply (4/4)

We can now assemble the entire FMM method for the HSS case. For the sake of simplicity we will assume that all leaf nodes are at level K.

$$
\begin{aligned}
g_{K;i} &= V_{K;i}^T \, x_{K;i} \\
g_{k;i} &= W_{k+1;2i}^T \, g_{k+1;2i} + W_{k+1;2i+1}^T \, g_{k+1;2i+1} \\
f_{0;0} &= [\,] \\
f_{k+1;2i} &= B_{k;2i,2i+1} \, g_{k;2i+1} + R_{k;2i} f_{k;i} \\
f_{k+1;2i+1} &= B_{k;2i+1,2i} \, g_{k;2i} + R_{k;2i+1} f_{k;i} \\
b_{K;i} &= A_{K;i,i} \, x_{K;i} + U_{K;i} f_{k;i}
\end{aligned}
$$

This algorithm costs $O(n)$ operations.

Multiply signal flow graph



Solver (1/5)

- We observe that the FMM recursions are linear. So we can encode them in a sparse matrix.
- Fix an **ordering** of the nodes, $(k;i)$, of the binary partition tree and let g denote the corresponding vector of the $g_{k;i}$'s. Similarly for f.
- Let $Z_\vee$ denote the down-shift matrix on the binary partition tree:

$$(Z_\vee f)_{k;i} = f_{k-1;\lfloor \frac{i}{2} \rfloor}$$

  Note that $(Z_\vee f)_{0;0} = 0$ and the values on the leaf, $f_{K;i}$, are lost after this operation.
- $Z_\vee^T$ denotes the *up-shift and add* operation on the binary partition tree:

$$(Z_\vee^T g)_{k;i} = g_{k+1;2i} + g_{k+1;2i+1}$$

  $Z_\vee^T$ introduces zeros into the leaves and looses the value at the root.

Solver (2/5)

- Let $Z_{\leftrightarrow}$ denote the *exchange-siblings* operator on the binary tree:

$$(Z_{\leftrightarrow} g)_{k;2i} = g_{k;2i+1} \qquad (Z_{\leftrightarrow} g)_{k;2i+1} = g_{k;2i}$$

- Let $\mathbf{P}_{\text{leaf}}$ denote the linear operator that *projects onto the leaves* of the binary partition tree:

$$(\mathbf{P}_{\text{leaf}}\, x)_{K;i} = x_{K;i} \qquad \text{otherwise} \qquad (\mathbf{P}_{\text{leaf}}\, x)_{k;i} = 0$$

  Note that

$$(\mathbf{P}_{\text{leaf}}^T f)_{K;i} = f_{K;i}$$

- Note that $Z_{\vee}\mathbf{P}_{\text{leaf}} = 0$.

Solver (3/5)

In the chosen order of the nodes of the binary partition tree define the following:

- W: block diagonal matrix of the $W_{k;i}$'s. Similarly for R.
- U: block diagonal matrix of the $U_{K;i}$'s. Similarly for V.
- B: block tri-diagonal matrix of the $B_{k;i,j}$'s.
- D: block diagonal matrix of the $A_{K;i,i}$'s.
- Now we can write the matrix–vector multiplication recursions as:

$$\begin{aligned} g &= Z_{\vee}^{T} W^{T} g + \mathbf{P}_{\text{leaf}} V^{T} x \\ f &= R Z_{\vee} f + B Z_{\leftrightarrow} g \\ b &= D x + U \mathbf{P}_{\text{leaf}}^{T} f \end{aligned}$$

Solver (4/5)

We can merge all these equations into a single sparse matrix:

$$\begin{bmatrix} I - Z_{\vee}^T W^T & 0 & -\mathbf{P}_{\text{leaf}} V^T \\ -B Z_{\leftrightarrow} & I - R Z_{\vee} & 0 \\ 0 & U \mathbf{P}_{\text{leaf}}^T & D \end{bmatrix} \begin{bmatrix} g \\ f \\ x \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ b \end{bmatrix}$$

Eliminating g and f we obtain the diagonal form of the HSS representation

$$A = D + U \mathbf{P}_{\text{leaf}}^T (I - R Z_{\vee})^{-1} B Z_{\leftrightarrow} (I - Z_{\vee}^T W^T)^{-1} \mathbf{P}_{\text{leaf}} V^T$$

Solver (5/5)

- However there is another elimination order that has essentially no fill-in.
- Observe that the graph correspondig to the above sparse matrix is the same as the signal flow graph for multiplication.
- The **depth-first ordering** of this graph is easily seen to be free of fill-in for both LU factorization and QR factorization.
- Therefore there is a numerically stable solver for HSS matrices that requires only $O(n)$ operations.
- One can also obtain explicit factorizations of HSS matrices where the factors are also in HSS form in $O(n)$ operations.

Multiplication (1/3)

Since

$$\begin{bmatrix} \clubsuit & A_{01} & \clubsuit \\ A_{10} & \clubsuit & A_{12} \\ \clubsuit & A_{21} & \clubsuit \end{bmatrix} \begin{bmatrix} \clubsuit & B_{01} & \clubsuit \\ B_{10} & \clubsuit & B_{12} \\ \clubsuit & B_{21} & \clubsuit \end{bmatrix} =$$

$$\begin{bmatrix} \clubsuit & \clubsuit B_{01} + A_{01}\clubsuit + \clubsuit B_{21} & \clubsuit \\ A_{10}\clubsuit + \clubsuit B_{10} + A_{12}\clubsuit & \clubsuit & A_{10}\clubsuit + \clubsuit B_{12} + A_{12}\clubsuit \\ \clubsuit & \clubsuit B_{01} + A_{21}\clubsuit + \clubsuit B_{21} & \clubsuit \end{bmatrix}$$

- Therefore the HSS-rank of AB is at most the sum of the HSS-ranks of A and B.
- The corresponding fast algorithm can be derived algebraically from the diagonal representation (unpublished).

Multiplication (2/3)

The recursions for $AB = C$:

$$
\begin{aligned}
G_{K;i} &= V_{K;i}^T(A)U_{K;i}(B) \\
G_{k;i} &= W_{k+1;2i}^T(A)G_{k+1;2i}R_{k+1;2i}(B) + \\
&\quad W_{k+1;2i+1}^T(A)G_{k+1;2i+1}R_{k+1;2i+1}(B) \\
F_{0;0} &= [\,] \\
F_{k+1;2i} &= B_{k+1;2i,2i+1}(A)G_{k+1;2i+1}B_{k+1;2i+1,2i}(B) \\
&\quad + R_{k+1;2i}(A)F_{k;i}W_{k+1;2i}^T(B) \\
D_{K;i} &= D_{K;i}(A)D_{K;i}(B) + U_{K;i}(A)F_{k;i}V_{K;i}^T(B) \\
U_{K;i} &= \begin{bmatrix} D_{K;i}(A)U_{K;i}(B) & U_{K;i}(A) \end{bmatrix} \\
V_{K;i} &= \begin{bmatrix} V_{K;i}(B) & D_{K;i}^T(B)V_{K;i}(A) \end{bmatrix}
\end{aligned}
$$

Multiplication (3/3)

The rest of the recursions for $AB = C$:

$$B_{k;2i,2i+1} = \begin{bmatrix} B_{k;2i,2i+1}(B) & 0 \\ R_{k;2i}(A)F_{k-1;i}W^T_{k;2i+1}(B) & B_{k;2i,2i+1}(A) \end{bmatrix}$$

$$B_{k;2i+1,2i} = \begin{bmatrix} B_{k;2i+1,2i}(B) & 0 \\ R_{k;2i+1}(A)F_{k-1;i}W^T_{k;2i}(B) & B_{k;2i+1,2i}(A) \end{bmatrix}$$

$$R_{k;2i} = \begin{bmatrix} R_{k;2i}(B) & 0 \\ B_{k;2i,2i+1}(A)G_{k;2i+1}R_{k;2i+1}(B) & R_{k;2i}(A) \end{bmatrix}$$

$$R_{k;2i+1} = \begin{bmatrix} R_{k;2i+1}(B) & 0 \\ B_{k;2i+1,2i}(A)G_{k;2i}R_{k;2i}(B) & R_{k;2i+1}(A) \end{bmatrix}$$

$$W_{k;2i} = \begin{bmatrix} W_{k;2i}(B) & B^T_{k;2i+1,2i}(B)G^T_{k;2i+1}W_{k;2i+1}(B) \\ 0 & W_{k;2i}(A) \end{bmatrix}$$

$$W_{k;2i+1} = \begin{bmatrix} W_{k;2i+1}(B) & B^T_{k;2i,2i+1}(B)G^T_{k;2i}W_{k;2i}(B) \\ 0 & W_{k;2i+1}(A) \end{bmatrix}$$

LU factorization (1/6)

From

$$\begin{bmatrix} \clubsuit & A_{01} & \clubsuit \\ A_{10} & \clubsuit & A_{12} \\ \clubsuit & A_{21} & \clubsuit \end{bmatrix} = \begin{bmatrix} I & 0 & 0 \\ A_{10}\clubsuit & I & 0 \\ \clubsuit & (A_{21} + \clubsuit A_{01})\clubsuit & I \end{bmatrix} \begin{bmatrix} \clubsuit & A_{01} & \clubsuit \\ 0 & \clubsuit & A_{12} + A_{10}\clubsuit \\ 0 & 0 & \clubsuit \end{bmatrix}$$

- Therefore the LU factors has the same HSS-rank as A.
- There is a fast algorithm to find the HSS representation of the LU factors from that of A.
- We use Gu's approach and factor

$$LU = A_{0;0,0} + U_{0;0} F_{0;0} V_{0;0}^T$$

- The derivation is a bit tricky so we just present the final recursions.

LU factorization (2/3)

$$
\begin{aligned}
F_{0;0} &= [\,] \\
F_{k;2i} &= R_{k;2i} F_{k-1;i} W_{k;2i}^T \\
B_{k;2i+1,2i}(L) &= B_{k;2i+1,2i} + R_{k;2i+1} F_{k-1;i} W_{k;2i}^T \\
B_{l;2i,2i+1}(U) &= B_{k;2i+1,2i} + R_{k;2i} F_{k-1;i} W_{k;2i+1}^T \\
D_{K;i}(L) D_{K;i}(U) &= D_{K;i} + U_{K:i} F_{K;i} V_{K;i}^T \\
U_{K;i}(U) &= D_{K;i}^{-1}(L) U_{K;i} \\
V_{K;i}(U) &= V_{K;i} \\
U_{K;i}(L) &= U_{K;i} \\
V_{K;i}(L) &= D_{K;i}^{-T}(U) V_{K;i}
\end{aligned}
$$

LU factorization (3/3)

$$
\begin{aligned}
G_{K;i} &= V_{K;i}^T(L)U_{K;i}(U) \\
G_{k-1;i} &= W_{k;2i}^T(L)G_{k;2i}R_{k;2i}(U) + W_{k;2i+1}^T(L)G_{k;2i+1}R_{k;2i+1}(U) \\
R_{k;2i}(U) &= R_{k;2i} \\
R_{k;2i+1}(U) &= R_{k;2i+1} - B_{k;2i+1,2i}(L)G_{k;2i}R_{k;2i} \\
W_{k;i}(U) &= W_{k;i} \\
R_{k;i}(L) &= R_{k;i} \\
W_{k;2i}(L) &= W_{k;2i} \\
W_{k;2i+1}(L) &= W_{k;2i+1} - B_{k;2i,2i+1}^T(L)G_{k;2i}R_{k;2i}
\end{aligned}
$$

Lower inverse (1/2)

- From

$$\begin{bmatrix} \clubsuit & 0 & 0 \\ A_{10} & \clubsuit & 0 \\ \clubsuit & A_{21} & \clubsuit \end{bmatrix}^{-1} = \begin{bmatrix} \clubsuit & 0 & 0 \\ \clubsuit A_{10} \clubsuit & \clubsuit & 0 \\ \clubsuit & \clubsuit A_{21} \clubsuit & \clubsuit \end{bmatrix}$$

  we see that the HSS-rank of the inverse of a lower triangular matrix is the same as that of the original matrix.
- There are recursions to compute the HSS representation of the inverse of a lower triangular in $O(n)$ operations from that of the lower triangular matrix.

Lower inverse (2/2)

$$
\begin{aligned}
G_{K;i} &= V_{K;i}^T D_{K;i}^{-1} U_{K;i} \\
G_{k-1;i} &= W_{k;2i}^T G_{k;2i} R_{k;2i} - \\
&\quad W_{k;2i+1}^T G_{k;2i+1} B_{k;2i+1,2i} G_{k;2i} R_{k;2i} + \\
&\quad W_{k;2i+1}^T G_{k;2i+1} R_{k;2i+1} \\
U_{K;i}(L^{-1}) &= D_{K;i}^{-1} U_{K;i} \\
V_{K;i}(L^{-1}) &= D_{K;i}^{-T} V_{K;i} \\
R_{k;2i}(L^{-1}) &= R_{k;2i} \\
R_{k;2i+1}(L^{-1}) &= -B_{k;2i+1;2i} G_{k;2i} R_{k;2i} + R_{k;2i+1} \\
W_{k;2i}(L^{-1}) &= W_{k;2i} - B_{k;2i+1,2i}^T G_{k;2i+1}^T W_{k;2i+1} \\
W_{k;2i+1}(L^{-1}) &= W_{k;2i+1}
\end{aligned}
$$

Inverse

Note that

$$\begin{bmatrix} \clubsuit & A_{01} & \clubsuit \\ A_{10} & \clubsuit & A_{12} \\ \clubsuit & A_{21} & \clubsuit \end{bmatrix}^{-1} = \begin{bmatrix} \clubsuit & (\clubsuit A_{01} + \clubsuit A_{21}) S_v & \clubsuit \\ S_h(A_{10}\clubsuit + A_{12}\clubsuit) & \clubsuit & S_h(A_{10}\clubsuit + A_{12}\clubsuit) \\ \clubsuit & (\clubsuit A_{01} + \clubsuit A_{21}) S_v & \clubsuit \end{bmatrix}$$

- So the HSS-rank of A^{-1} is the identical to the HSS-rank of A.
- We can assemble a fast algorithm for computing the HSS representation of A^{-1} from that of A. Compute:
  - LU factorization of A in HSS form;
  - the HSS form of L^{-1} and U^{-1};
  - the HSS form of $U^{-1}L^{-1} = A^{-1}$.

Concluding remarks

- Displacement structure
- SSS
- HSS
- General

Displacement structure

- Given a family of matrices does it have a **useable** displacement structure?
- Given a fast displacement structured family can we characterize the family directly?
- Can we generalize displacement structure to sums of tensor products?
- What is special about the eigenvectors?
- Is there a general approach fast matrix–vector multiplies?
- Is there a general approach to super-fast algorithms?

SSS

- Very systematic theory that traces its roots back to complex analysis.
- Can we generalize it?
- See my SIAM ALA'18 talk on our web site.
- Structure of eigenvectors?

HSS

- Coherent derivation via sparse matrices (similar to SSS) for matrix–vector multiply and linear system solution.
- Ad hoc methods for matrix–matrix multiply and LU factorization. Can we do better?
- Closely related to the more general FMM representation.
- FMM representations change under multiplication which seems to rule out fast **exact** factorization and inverse.
- See my afore mentioned SIAM ALA'18 talk on what we can potentially do.

General

- The big question in practice is tied closely to the exact inverse of Cauchy-like matrices where the points are drawn from higher dimensions.
- These slides can be downloaded from our web-site `http://scg.ece.ucsb.edu`

Thank You!

References

[1] Shivkumar Chandrasekaran, Patrick Dewilde, Ming Gu, T Pals, Xiaorui Sun, Alle-Jan van der Veen, and Daniel White. Some fast algorithms for sequentially semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 27(2):341–364, 2005.

[2] Shivkumar Chandrasekaran, Ming Gu, and Timothy Pals. A fast ulv decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622, 2006.

[3] Shivkumar Chandrasekaran and Ali H Sayed. Stabilizing the generalized schur algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):950–983, 1996.

[4] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[5] Patrick Dewilde and Alle-Jan van der Veen. *Time-Varying Systems and Computations*. Springer Science & Business Media, 1998.

[6] Israel Gohberg, Thomas Kailath, and Vadim Olshevsky. Fast gaussian elimination with partial pivoting for matrices with displacement structure. *Mathematics of Computation*, 64(212):1557–1576, 1995.

[7] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.*, 73:325–348, 1987.

[8] Thomas Kailath, Sun-Yuan Kung, and Martin Morf. Displacement ranks of matrices and linear equations. *Journal of Mathematical Analysis and Applications*, 68(2):395–407, 1979.

[9] V. Strassen. Gaussian Elimination is Not Optimal. *Numerische Mathematik*, 13:354–356, 1969.