3 *Structured algorithms for stuctured matrices*

Reading material FFT: Björck Numerical methods in matrix computations, 2014. chapter 1.8.5

3.1 *Fast fourier transform*

The discrete Fourier transform is defined as follows. Let $f(x)$ be a function, with known function values at $f_k = f(x_k)$, $k = 0, \dots, N-1$. Then, we can view the transform as an interpolation in the points $x_0, \dots, x_{N-1}$

$$x_k = \frac{2\pi k}{N} \tag{3.1}$$

Note that we assume that the grid is (equispaced) uniform in (3.1). The techniques for the main algorithms are not trivially transferred to the non-uniform case. Non-uniform FFT is an active research area.

with the exponential as basis functions:

$$f^*(x) = \sum_{j=0}^{N-1} c_j e^{ijx}$$

The coefficients $c_0, \dots, c_{N-1}$ are given by

$$c_j = \frac{1}{N} \sum_{k=0}^{N-1} f_k e^{-ijx_k}.$$

We now let ω_N be the Nth root of unity

$$\omega_N = e^{-2\pi i/N}. \tag{3.2}$$

The coefficients can also be expressed as

$$c_j = \frac{1}{N} \sum_{k=0}^{N-1} f_k \omega_N^{jk}. \tag{3.3}$$

The naive approach to compute the coefficients c_j would require $\mathcal{O}(N^2)$ operations. The FFT is a procedure which achieves this in $\mathcal{O}(N\log(N))$ operations. To ease the notation we define $y_i = Nc_i$, such that the factor $1/N$ in (3.3) can be removed.

The discrete Fourier transform (DFT) is defined from the DFT matrix.

Definition 3.1.1 (DFT and DFT matrix)**.** *The application of DFT to the vector $f \in \mathbb{C}^N$ is*

$$y = F_N f$$

where

$$F_N = \begin{bmatrix} \omega_N^{00} & \cdots & \omega_N^{0(N-1)} \\ \vdots & & \vdots \\ \omega_N^{(N-1)0} & \cdots & \omega_N^{(N-1)(N-1)} \end{bmatrix} \tag{3.4}$$

and ω_N *is given by* (3.2).

It is easy to verify that

$$\frac{1}{N} F_N^H F_N = I$$

which implies that the inverse DFT transform satisfies

$$f = \frac{1}{N} F_N^H y$$

The hermitian transpose of the matrix is the same as the matrix (3.4), if we replace ω_N with $\bar{\omega}_N = e^{2\pi i/N}$. Therefore, the algorithm developed below can analogously be applied to compute the inverse DFT.

The approach that follows assumes that the size of the matrix is $N = 2^p$. This is not a dramatic assumption in practice, since the matrix size can be increased by adding trivial equations. Let $m = N/2$. We now reorder the equations, by considering odd and even parts separately. For $j = 0, \ldots, N-1$,

$$y_j = \sum_{k_1=0}^{m-1} (\omega_N^2)^{jk_1} f_{2k_1} + \omega_N^j \sum_{k_1=0}^{m-1} (\omega^2)^{jk_1} f_{2k_1+1}. \tag{3.5}$$

Now let $j = \beta m + j_1$, where j_1 is the remainder in the division j/m, such that

$$(\omega_N^2)^{jk_1} = \omega_m^{j_1 k_1}.$$

We can now define

$$\phi_{j_1} := \sum_{k_1=0}^{m-1} \omega_m^{j_1 k_1} f_{2k_1} \quad \text{and} \quad \psi_{j_1} := \sum_{k_1=0}^{m-1} \omega_m^{j_1 k_1} f_{2k_1+1} \tag{3.6}$$

The equations (3.6) is again an FFT-problem of half the size, and the solution can be constructed from (3.7).

and (3.5) can be reduced to $y_j = \phi_{j_1} + \omega_N^j \psi_{j_1}$.

The relations (3.7) are called the butterfly relations.

$$y_{j_1} = \phi_{j_1} + \omega_N^j \psi_{j_1} \tag{3.7a}$$

$$y_{j_1+N/2} = \phi_{j_1} - \omega_N^j \psi_{j_1} \tag{3.7b}$$

Note that the equations (3.6) are two fourier series which allows us to repeat the procedure on a smaller series. The recursion can be repeated until we obtain a small fourier series which can be computed directly.

A general implementation for matrices of size $N = 2^p$ is given in Algorithm 1.

```
function fftx(x);
Input: x
Output: y
Input: The matrix A ∈ R^{n×n} and vector b.
n =length (x)
omega = exp(-2i*pi/n);
if rem(n,2)=0 then
    k=(0:n/2-1)'; w=omega.^k;
    u=fftx(x(1:2:n-1));
    v=w.*fftx(x(2:2:n));
    y=[u+v;u-v];
else
    j=0:n-1; k=j';
    F=omega.^(k*j);
    y=F*x;
end
```

Algorithm 1: Recursive formulation of FFT. Algorithm 1.8.1 in Björck

Reading material: Cooley-Tukey FFT: Björck: pages 194-196.

3.2 *Toeplitz matrices*

Reading material: Golub & Van Loan, chapter 4.7

Levinson-Durbin: Golub and van Loan

3.3 *Circulant matrices*

Reading material: Golub & Van Loan, chapter 4.8

3.4 *Hierarchical and semi-separable matrices*

3.4.1 *Semi-separable matrices*

Many matrices stemming from data have some form of low-rank structure. The semi-separable is a low-rank structure where involving the lower triangular part.

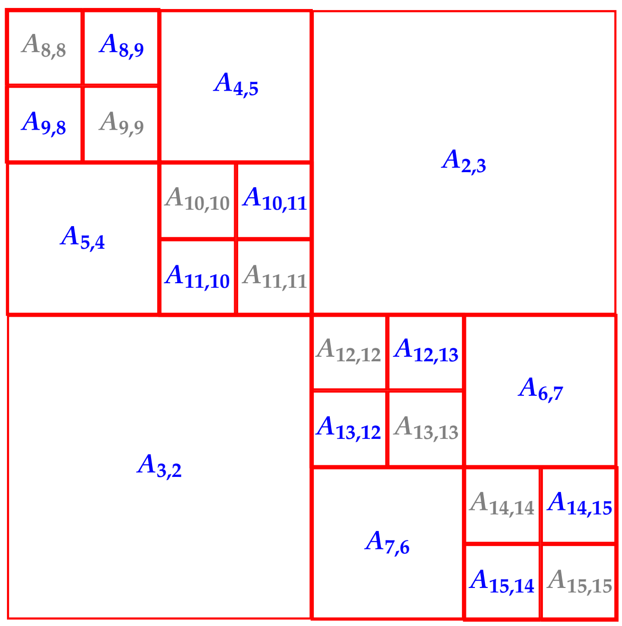
Definition 3.4.1. *A symmetric matrix is called semi-separable (of order p), if all submatrices taken out of the low triangular part have rank $\leq p$.*

Example: Banded matrices is semi-separable with order same as the band

Example: The inverse of a tridiagonal matrix is semi-separable
Example: Electrostatics
Properties: Inverse of a semi-separable matrix is again semi-separable.

3.4.2 *Hierarchical semi-separable matrices*

We will now the exploit the properties of semi-separable matrices, considering a particular type of partitioning of the matrix.



What follows will be based on two observations.

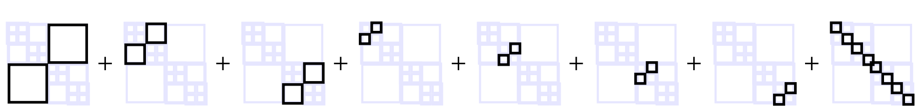
The system for the number of the blocks is explained by the recursive subdivision formulation in Section 3.4.3.

1. Due to the property that the matrix is semi-separable, all the matrices below the diagonal will be of low-rank. Therefore, the blocks in the the nodes marked in blue above are of low-rank. We assume those blocks are given in terms of SVD-factorizations:

   $$A_{\sigma,\tau} = U_\sigma \tilde{A}_{\sigma,\tau} V_\tau^T.$$

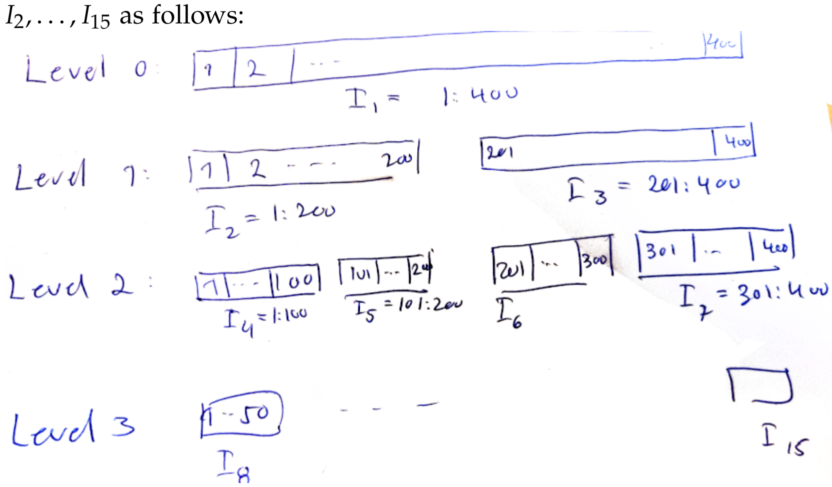
   The matrix $\tilde{A}_{\sigma,\tau}$ in the middle of the factorization can be (but does not have to be) a diagonal matrix and is called the **core** of the block.

2. In order to implement operations with this matrix, we separate the matrix into a sum of blocks. We will loop through all the blocks of the matrix in the following way. Note that the final matrix is just a diagonal matrix.



3.4.3 *Recursive index vector formulation*

In order to exploit the second observation, we will formulate it in a recursive way in terms of index vectors. First, define the index vectors $I_1, I_2, \ldots, I_{15}$ as follows:

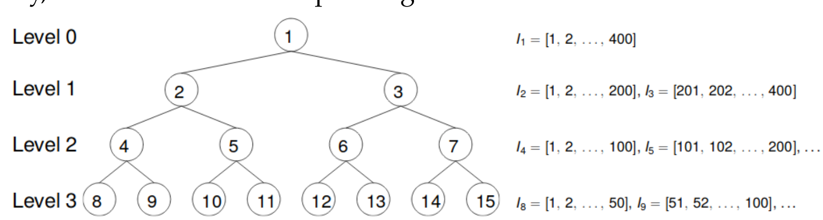


With this notation, we can denote the blocks of the matrix as

$$A_{\sigma,\tau} = A(I_\sigma, I_\tau), \quad \sigma, \tau = 1, \ldots, 15.$$

Credit for the image: PGM. Will be replaced.

Algorithms for this matrix are naturally formulated in a recursive way, so we draw the corresponding tree:



3.4.4 *A recursive algorithm for the matrix vector product Ac*

The above tools can be used to carry out several operations associated with A. Most importantly, we now show how it can be combined into an algorithm to compute the matrix times a vector c.

** Matrix-vector multiply using tree and index vectors **

loop τ is a node in the tree
  if τ is a leaf
    $b(I_\tau) = b(I_\tau) + A_{\tau,\tau} c(I_\tau)$
  else
    Let σ_1 and σ_2 be children of τ
    Compute the following by a recursive call

$$b(I_\tau) = b(I_\tau) + \begin{bmatrix} 0 & A_{\sigma_1,\sigma_2} \\ A_{\sigma_2,\sigma_1} & 0 \end{bmatrix} \begin{bmatrix} c(I_{\sigma_1}) \\ c(I_{\sigma_2}) \end{bmatrix} \tag{3.8}$$

Note that the matrix product in (3.8) can be computed with computation of $A_{\sigma_1,\sigma_2} c(\sigma_2)$ and $A_{\sigma_2,\sigma_1} c(\sigma_1)$, which can be computed by two recursive calls.


```
    end
end
```

3.5 *Other structures*