A FASTER MULTIPOLE LEGENDRE-CHEBYSHEV TRANSFORM

MIKAEL MORTENSEN*

Abstract. This paper describes a fast algorithm for transforming Legendre coefficients into Chebyshev coefficients, and vice versa. The algorithm is based on the fast multipole method and is similar to the approach described by Alpert and Rokhlin [SIAM J. Sci. Comput., 12 (1991)]. The main difference is that we utilise a modal Galerkin approach with Chebyshev basis functions instead of a nodal approach with a Lagrange basis. Part of the algorithm is a novel method that facilitates faster spreading of intermediate results through neighbouring levels of hierarchical matrices. This enhancement leads to a method that is approximately $10-20$ % faster to execute, contingent on chosen parameters. We also describe an efficient initialization algorithm that for the Lagrange basis is roughly 5 times faster than the original method for large input arrays. The described method has both a planning and execution stage that asymptotically require $\mathcal{O}(N)$ flops. The algorithm is simple enough that it can be implemented in 100 lines of vectorized Python code. Moreover, its efficiency is such that a more optimized C implementation can transform 10^6 coefficients in approximately 40 milliseconds on a MacBook Pro, representing about 3 times the execution time of a well-planned type 2 discrete cosine transform from FFTW (www.fftw.org). Planning for the 10^6 coefficients requires approximately 70 milliseconds.

1. Introduction.

Let $\mathbb{P}_N$ be the set of polynomials of degree less than or equal to N. Any function $f(x) \in \mathbb{P}_{N-1}$ defined on $x \in [-1,1]$ can then be expanded as

$$f(x) = \sum_{j=0}^{N-1} \hat{f}_j^{leg} L_j(x) = \sum_{j=0}^{N-1} \hat{f}_j^{cheb} T_j(x), \tag{1.1}$$

where $\{L_j\}_{j=0}^{N-1}$ and $\{T_j\}_{j=0}^{N-1}$ are the first N Legendre polynomials and Chebyshev polynomials of the first kind, respectively, and $\boldsymbol{f}^{leg} = \{\hat{f}_j^{leg}\}_{j=0}^{N-1}$ and $\boldsymbol{f}^{cheb} = \{\hat{f}_j^{cheb}\}_{j=0}^{N-1}$ are expansion coefficients. It is the purpose of this paper to describe a fast method for computing $\boldsymbol{f}^{cheb}$ from $\boldsymbol{f}^{leg}$ and vice versa.

A direct method can be obtained by using the orthogonality of Chebyshev polynomials in the $L^2_\omega(-1,1)$ space, defined with the weighted inner product

$$(f,v)_\omega = \int_{-1}^{1} f v \omega dx, \tag{1.2}$$

where the weight $\omega(x) = (1-x^2)^{-1/2}$ and $f(x)$ and $v(x)$ are real functions. We multiply the two equal sums in (1.1) by the test function T_i and the weight ω and integrate over the domain to obtain

$$\sum_{j=0}^{N-1} (T_i, L_j)_\omega \hat{f}_j^{leg} = \sum_{j=0}^{N-1} (T_i, T_j)_\omega \hat{f}_j^{cheb}, \quad i = 0, 1, \ldots, N-1. \tag{1.3}$$

Here $(T_i, T_j)_\omega = c_i \pi/2 \delta_{ij}$, where $c_0 = 2$, $c_i = 1$ for $i > 0$, δ_{ij} is the Kronecker delta-function and the diagonal matrix is denoted as $\boldsymbol{C} = \text{diag}(\{c_i\pi/2\}_{i=0}^{N-1}) \in \mathbb{R}^{N\times N}$. The matrix on the left hand side is denoted as $a_{ij} = (T_i, L_j)_\omega$, with $\boldsymbol{A} = (a_{ij})_{i,j=0}^{N-1} \in \mathbb{R}^{N\times N}$, such that

$$\boldsymbol{A}\boldsymbol{f}^{leg} = \boldsymbol{C}\boldsymbol{f}^{cheb}, \tag{1.4}$$

*Department of Mathematics, University of Oslo, Norway (mikaem@math.uio.no).

and thus

$$\boldsymbol{f}^{cheb} = \boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{f}^{leg} \quad \text{and} \quad \boldsymbol{f}^{leg} = \boldsymbol{A}^{-1}\boldsymbol{C}\boldsymbol{f}^{cheb}. \tag{1.5}$$

The diagonal matrix $\boldsymbol{C}$ is obviously trivial to apply to any vector, and thus we will focus on the matrix $\boldsymbol{A}$, where all nonzero elements can be represented as

$$a_{ij} = \Lambda\left(\frac{j-i}{2}\right)\Lambda\left(\frac{j+i}{2}\right), \tag{1.6}$$

where

$$\Lambda(x) = \frac{\Gamma(x+\frac{1}{2})}{\Gamma(x+1)}, \quad x \in \mathbb{R}^+. \tag{1.7}$$

The upper triangular matrix $\boldsymbol{A}$ can be even more easily represented diagonalwise, where the only nonzero items are

$$a_{i,i+2k} = \Lambda(k)\Lambda(i+k), \quad \forall\, k = 0, 1, \ldots, N/2-1 \text{ and } i+k < N. \tag{1.8}$$

Hence, if we define a vector $\boldsymbol{\lambda} = \{\lambda_k\}_{k=0}^{N-1} \in \mathbb{R}^N$, where $\lambda_k = \Lambda(k)$, then the entire matrix $\boldsymbol{A}$ can be represented using merely a single vector $\boldsymbol{\lambda}$ of storage. Also note that since every other upper diagonal is zero, the matrix can be efficiently split into odd and even parts. A direct algorithm using $\boldsymbol{A}$ in Eq. (1.5) is briefly described in Sec. 2.

The transforms between $\boldsymbol{f}^{leg}$ and $\boldsymbol{f}^{cheb}$ are useful because, although Chebyshev and Legendre expansions share similar approximation characteristics, each offers distinct advantages and drawbacks. A Chebyshev series is orthogonal in the weighted $L^2_\omega(-1,1)$ space, and it is often favoured because algorithms can be accelerated through the use of cosines and fast discrete cosine transforms. The Legendre polynomials, on the other hand, are often favoured for their orthogonality in the more convenient $L^2(-1,1)$ space. However, the Legendre polynomials lack really fast transforms, which makes them slow to evaluate on a large physical mesh. The Chebyshev-Legendre Galerkin method of Shen [15] takes advantage of the best of both worlds, making use of a transform between $\boldsymbol{f}^{leg}$ and $\boldsymbol{f}^{cheb}$, combined with fast cosine transforms.

There are several methods available for transforming from $\boldsymbol{f}^{leg}$ to $\boldsymbol{f}^{cheb}$ and vice versa. To date the state-of-the-art in widely recognised to be the fast multipole method described by Alpert and Rokhlin [2], which completes the transform in $\mathcal{O}(N)$ operations after an initialization stage that is also of $\mathcal{O}(N)$. This method makes use of the analytical and low-rank property of $\boldsymbol{A}$ as well as hierarchical matrices, and it is the method that we will attempt to improve in Sec. 4. Keiner [8] extends this method to transforms between families of Gegenbauer polynomials and improves the error estimates of the approximations. Shen et al. [16] describe a similar method for Jacobi polynomials, also based on the low-rank property of the connection matrices and using hierarchical semiseparable matrices. The algorithm of Shen et al. also requires $\mathcal{O}(N)$ operations for evaluation after a longer $\mathcal{O}(N^2)$ initialization stage. Keiner [9] also makes use of semiseparable matrices and describes an $\mathcal{O}(N \log N)$ divide and conquer approach. Hale and Townsend [7] describe a $\mathcal{O}(N(\log N)^2/\log\log N)$ method based on Stieltjes' asymptotic formula for Legendre polynomials [19]. This method is sold on the merit of requiring no initialization (except from the FFTs involved) and being simple enough to be implemented in 100 lines of MATLAB code. An even simpler $\mathcal{O}(N \log^2 N)$ approach is described by Townsend et al. [20], decomposing the

connection matrix into diagonally scaled Hadamard products involving Toeplitz and Hankel matrices. This method is implemented by the MATLAB software system Chebfun [4] and the Julia package called FastTransforms.jl [18]. The method to be described in Sec. 4 will be shown to be significantly faster and can also be implemented in approximately 100 lines of compact Python code.

2. A direct method. The structure of $\boldsymbol{A}$ (see Eqs. (1.8) and (2.1)) allows for a low memory and quick direct approach for computing $\boldsymbol{A}\boldsymbol{f}^{leg}$ and thus $\boldsymbol{f}^{cheb}$ from $\boldsymbol{f}^{leg}$. The method applies diagonalwise as shown in Alg. 2.1. The method is very efficient for small N, but naturally it will become expensive for large N since the number of floating point operations (flops) of a direct matrix vector product scales like $\mathcal{O}(N^2)$. We note that a direct method that precomputes all $\lambda_{n/2}\lambda_{n/2+i}$ (see Alg. 2.1) will lead to fewer floating point operations for the execution, but it is not necessarily faster, because the alternative leads to fewer memory access operations. To see this consider what the matrix $\boldsymbol{A}$ looks like in Eq. (2.1):

$$\boldsymbol{A} = \begin{bmatrix} \lambda_0\lambda_0 & 0 & \lambda_1\lambda_1 & 0 & \lambda_2\lambda_2 & 0 & \cdots \\ 0 & \lambda_0\lambda_1 & 0 & \lambda_1\lambda_2 & 0 & \lambda_2\lambda_3 & \cdots \\ 0 & 0 & \lambda_0\lambda_2 & 0 & \lambda_1\lambda_3 & 0 & \cdots \\ 0 & 0 & 0 & \lambda_0\lambda_3 & 0 & \lambda_1\lambda_4 & \cdots \\ 0 & 0 & 0 & 0 & \lambda_0\lambda_4 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & \lambda_0\lambda_5 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{2.1}$$

In order to compute with the 12 nonzero items of $\boldsymbol{A}$ shown, it is sufficient for the compiler to retrieve only 6 numbers from memory $(\lambda_0, \lambda_1, \ldots, \lambda_5)$. So, compared to precomputing all matrix items, memory access for the matrix is halved (for these items), whereas the flop count is increased from two (one multiplication and one addition) to three (two multiplications and one addition) per non-zero item. Whichever is faster needs to be tested on a given computing system.

Algorithm 2.1 A direct Legendre-to-Chebyshev transform $\boldsymbol{f}^{cheb} = \boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{f}^{leg}$

function DIRECTL2C($\boldsymbol{f}^{leg}, \boldsymbol{\lambda}, \boldsymbol{C}$)
input $\boldsymbol{f}^{leg}$: array $\in \mathbb{R}^N$ ▷ Legendre coefficients
input $\boldsymbol{\lambda}$: array $\in \mathbb{R}^N$ ▷ $\lambda_k = \Lambda(k)$, see Eq. (1.7)
input $\boldsymbol{C}$: matrix $\in \mathbb{R}^{N\times N}$ ▷ diagonal matrix, see Eq. (1.4)
output $\boldsymbol{f}^{cheb}$: array $\in \mathbb{R}^N$ ▷ Chebyshev coefficients
    $N \leftarrow$ **len** $\boldsymbol{f}^{leg}$
    $f^{cheb} \leftarrow 0$
    $n \leftarrow 0$
    while $n < N$ **do**
        for $i \leftarrow 0, N-n-1$ **do**
            $f_i^{cheb} \leftarrow f_i^{cheb} + \lambda_{n/2}\lambda_{n/2+i} f_{i+n}^{leg}$
        end for
        $n \leftarrow n+2$
    end while
    $\boldsymbol{f}^{cheb} \leftarrow \boldsymbol{C}^{-1}\boldsymbol{f}^{cheb}$
    return $\boldsymbol{f}^{cheb}$
end function

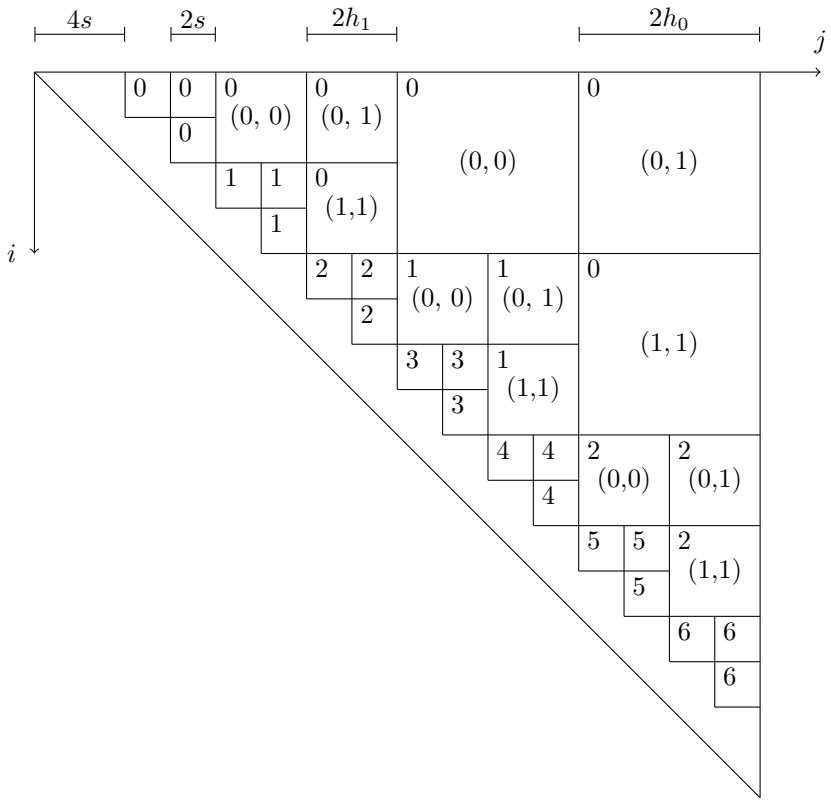


Fig. 1: Example of a decomposition of the matrix $\boldsymbol{A}$ using three levels. Level 0 is the one with the fewest blocks, and then levels 1 and 2 have gradually more. For levels 0 and 1 the local block numbers are given in the upper left corner of each square, whereas the local indices for a submatrix on a block are shown in parenthesis. For level 2 only the local block number is printed.

3. A Fast Multipole Method. We will first describe a fast $\mathcal{O}(N \log_2 N)$ multipole method for the matrix vector product $\boldsymbol{A}\boldsymbol{f}^{leg}$ that is very easy to implement.[1] The method uses the low-rank properties of $\boldsymbol{A}$ and the general idea from [2]. However, where [2] approximates $\boldsymbol{A}$ using nodal Lagrange polynomials in real space, we will take a modal approach and make use of Chebyshev polynomials in spectral space. In an effort to simplify notation, we will in this section write the matrix vector product as

$$\boldsymbol{z} = \boldsymbol{A}\boldsymbol{f}. \tag{3.1}$$

3.1. Decomposition into levels, blocks and square submatrices. The main idea of the fast multipole method is to decompose the matrix $\boldsymbol{A}$ into smaller submatrices, and then use low-rank approximations to efficiently compute the matrix-vector product of each submatrix. For decomposition of $\boldsymbol{A}$ we will use a system based on *levels*, *blocks* and square *submatrices*. Figure 1 shows an example where the upper

[1] The method for the reverse direction is basically identical and not shown in detail.

triangular part of $\boldsymbol{A}$ has been decomposed using three levels, i.e., there are square submatrices of exactly three different sizes. The largest squares represent level 0, and the two gradually smaller are thus levels 1 and 2. We will use $\gamma \in \mathbb{N}$ to refer to a specific level and the total number of levels is $L \in \mathbb{N}$. The structured decomposition of the matrix into levels, blocks and submatrices will allow us to easily create iterators that runs over all blocks, submatrices and vectors on any or all levels. Note that the unmarked part of the matrix closest to the main diagonal in Fig. 1 will be treated with a direct approach, as described in Sec 2.

On any given level we define a *block* as the three neighbouring *submatrices* of the same shape that share at least one common edge with at least one other submatrix in the block. The local (to each level) block numbers are given in Fig. 1 in the upper left corner of each square, and for level 2 the local block numbers are the only numbers printed. There are 11 blocks altogether in Fig. 1, as we are counting 1, 3 and 7 blocks on levels 0, 1 and 2, respectively.

All square submatrices on level γ have shape $2h_\gamma \times 2h_\gamma$, as shown in Fig. 1, where the even shape $2h_\gamma$ has been chosen since the matrix will be further decomposed into equally shaped odd and even parts. The size of the squares on each level, h_γ, can be computed as

$$h_\gamma = s\, 2^{L-\gamma-1}, \tag{3.2}$$

where $s = h_{L-1}$ is half the size of the smallest matrices, which needs to be specified.

The numbers in parenthesis in Fig. 1, in the center of the squares on levels 0 and 1, show the local indices $(p, q) \in \mathbb{N}^2$ used for a square *submatrix* on a block. For efficient storage and lookup these two indices will be mapped into a single index r using a column-major numbering scheme

$$r(p, q) = p + q(q + 1)/2, \quad r \in \{0, 1, 2\} \text{ and } q \geq p, \tag{3.3}$$

such that $r = 0, 1$ and $2 \implies (p, q) = (0, 0), (0, 1)$ and $(1, 1)$, respectively.

We access a submatrix using the level number γ, local block number b and the local submatrix index $r(p, q)$

$$\boldsymbol{A}(\gamma, b, r) = (a_{i+m,j+n})_{m,n=0}^{2h_\gamma-1} \in \mathbb{R}^{2h_\gamma \times 2h_\gamma}, \tag{3.4}$$

where the integers i and j represent the global indices of the upper left corner of the submatrix $\boldsymbol{A}(\gamma, b, r)$. It is easily shown that these global indices can be computed as

$$i(\gamma, b, p) = 2h_\gamma(2b + p) \in \mathbb{N} \text{ and } j(\gamma, b, q) = 2h_\gamma(2b + q + 2) \in \mathbb{N}. \tag{3.5}$$

Similarly, the odd ($\sigma = 1$) and even ($\sigma = 0$) parities of the same submatrix will be accessed as

$$\boldsymbol{A}^\sigma(\gamma, b, r) = (a_{i+2m+\sigma,j+2n+\sigma})_{m,n=0}^{h_\gamma-1} \in \mathbb{R}^{h_\gamma \times h_\gamma}. \tag{3.6}$$

The implementation of the FMM is based on an iterator that runs over all levels, blocks and submatrices in an ordered fashion. In order to create such an iterator we simply need to know the number of levels L and the number of blocks on each level b_γ. The number of blocks on a given level γ can be computed as twice the number of blocks on level $\gamma - 1$ plus one. Starting from $b_0 = 1$, this becomes through simple recursion

$$b_\gamma = 2^{\gamma+1} - 1. \tag{3.7}$$

Note that the chosen decomposition restricts the possible shapes N of the input vector $\boldsymbol{f} \in \mathbb{R}^N$, because the algorithm requires that

$$N = 2s(1 + b_L) = s2^{L+2}. \tag{3.8}$$

We may use Eq. (3.8) to compute the number of levels required by any input vector $\boldsymbol{f}$ of length $|\boldsymbol{f}|$ not necessarily satisfying (3.8)

$$L = \lceil \log_2 \frac{|\boldsymbol{f}|}{\hat{s}} \rceil - 2. \tag{3.9}$$

Here $\hat{s}$ is a chosen (optimal) size of the smallest submatrices and $\lceil x \rceil$ represents the ceiling function mapping x to the nearest integer greater than or equal to x. We then recompute $s = \lceil |\boldsymbol{f}|/2^{L+2} \rceil \in [\hat{s}/2, \hat{s}]$ and finally set $N = s2^{L+2}$, where $N \geq |\boldsymbol{f}|$. The bottom line is that any input array $\boldsymbol{f}$ needs to be padded with $s2^{L+2} - |\boldsymbol{f}|$ zeros for the algorithm to work. In what follows we will simply assume that N satisfies Eq. (3.8) and thus $L = \log_2 \frac{N}{s} - 2$.

The total number of blocks N_b and submatrices N_c are obtained by summing b_γ over all levels, and we obtain

$$N_b = \frac{N}{2s} - L - 2 \quad \text{and} \quad N_c = 3N_b, \tag{3.10}$$

respectively. Since the number of levels L grows as $\mathcal{O}(\log_2 N)$, we see that both the number of blocks and submatrices in Eq. (3.10) grow asymptotically as $\mathcal{O}(N)$.

The global vectors $\boldsymbol{z} \in \mathbb{R}^N$ and $\boldsymbol{f} \in \mathbb{R}^N$, that take part in the matrix vector product (3.1), are also looked up locally using the global indices i and j from Eq. (3.5). As such, we can easily find the parts of the global vectors that belong to any given level γ, block b and submatrix $r(p, q)$

$$\boldsymbol{f}(\gamma, b, q) = \{f_{j+m}\}_{m=0}^{2h_\gamma - 1} \in \mathbb{R}^{2h_\gamma},\ \boldsymbol{z}(\gamma, b, p) = \{z_{i+m}\}_{m=0}^{2h_\gamma - 1} \in \mathbb{R}^{2h_\gamma}. \tag{3.11}$$

Likewise, we get the larger vector blocks associated with entire blocks or levels as

$$\boldsymbol{f}(\gamma, b) = \{f_{j_b+m}\}_{m=0}^{4h_\gamma - 1} \in \mathbb{R}^{2\times 2h_\gamma},\ \boldsymbol{f}(\gamma) = \{f_{4h_\gamma+m}\}_{m=0}^{N-4h_\gamma - 1} \in \mathbb{R}^{b_\gamma \times 2\times 2h_\gamma}, \tag{3.12}$$

$$\boldsymbol{z}(\gamma, b) = \{z_{i_b+m}\}_{m=0}^{4h_\gamma - 1} \in \mathbb{R}^{2\times 2h_\gamma},\ \boldsymbol{z}(\gamma) = \{z_m\}_{m=0}^{N-4h_\gamma - 1} \in \mathbb{R}^{b_\gamma \times 2\times 2h_\gamma}, \tag{3.13}$$

where $i_b(b, \gamma) = 4bh_\gamma$ and $j_b(b, \gamma) = 4(1 + b)h_\gamma$ represent the indices of the upper left corner of block b on level γ. We treat the arrays in (3.12) and (3.13) as row-major, multidimensional arrays simply for convenience in future algorithms. However, the underlying data are just a single, contiguous block of numbers, and we might as well have been using, e.g., $\boldsymbol{f}(\gamma) \in \mathbb{R}^{4b_\gamma h_\gamma}$. For all vector blocks, if the vector is also endowed with a σ superscript, then a subdivision is further made into odd and even parts, like $\boldsymbol{f}^\sigma(\gamma, b, q) = \{f_{j+2m+\sigma}\}_{m=0}^{h_\gamma - 1} \in \mathbb{R}^{h_\gamma}$.

Figure 2 gives a more detailed and generic description of a zoomed-in part of a larger global matrix $\boldsymbol{A}$, illuminating how the global vectors on different levels are related. A correlation between vector blocks on neighbouring levels is obviously

$$\boldsymbol{f}(\gamma, b, q) = \boldsymbol{f}(\gamma + 1, 2b + q + 1), \tag{3.14}$$

$$\boldsymbol{z}(\gamma, b, p) = \boldsymbol{z}(\gamma + 1, 2b + p), \tag{3.15}$$

where the equalities should be understood in terms of the underlying data.

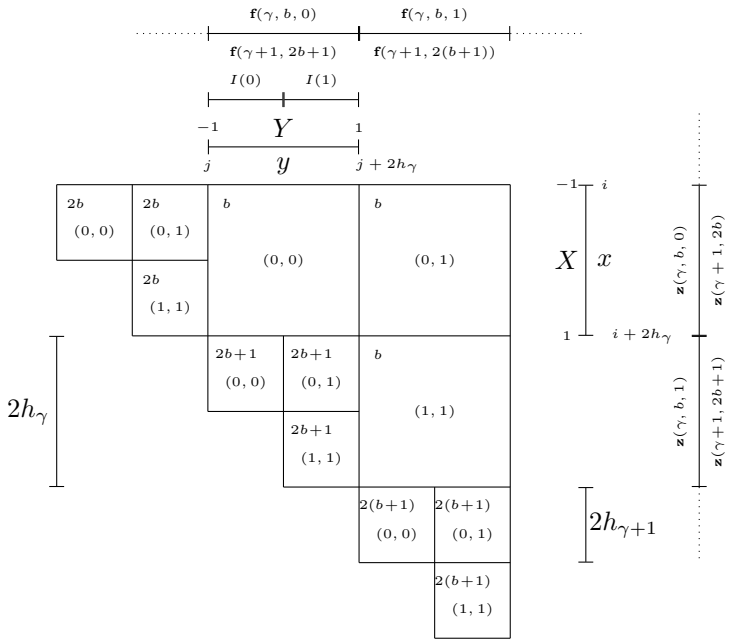


Fig. 2: Example of a local part of the matrix $\boldsymbol{A}$, showing two neighbouring levels. There is one block on level γ, with block number b. The block numbers on level $\gamma+1$ are shown in upper left corners as computed from b. Also shown are the reference domains X and Y on level γ. For submatrix $(0,0)$ on level γ, where the upper left corner is at global indices i, j, we also show the subintervals $I(0)$ and $I(1)$ used in Eq. (4.3).

3.2. The 2D forward Chebyshev transform. The most important part of the FMM is to approximate a submatrix $\boldsymbol{A}(\gamma, b, r)$ with a Chebyshev series. As described by Alpert and Rokhlin [2], we may view the global matrix $\boldsymbol{A}$ as a continuous function $\mathcal{A}(x, y) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, such that $\mathcal{A}(i, j) = a_{ij}$ for all nonzero items of $\boldsymbol{A}$. The continuous function

$$\mathcal{A}(x, y) = \Lambda\left(\frac{y-x}{2}\right)\Lambda\left(\frac{y+x}{2}\right), \tag{3.16}$$

is smooth for $y > x$.

For all square submatrices $\boldsymbol{A}(\gamma, b, r)$ we will approximate $\mathcal{A}(x, y)$ in the corresponding domain $\Omega(\gamma, b, r) = [i, i + 2h_\gamma] \times [j, j + 2h_\gamma]$, by projection to a two-dimensional reference tensor product space $W = V_M(I) \otimes V_M(I)$, where $I = [-1, 1]$, $V_M(I) = \text{span}\{T_k(X)\}_{k=0}^{M-1}$ and $X \in I$ is a reference coordinate. For any square submatrix with upper left corner in (i, j), there is an affine mapping from the physical coordinates $(x, y) \in \Omega$ to the reference coordinates

$$X = -1 + (x - i)/h_\gamma \text{ and } Y = -1 + (y - j)/h_\gamma, \tag{3.17}$$

where (i, j) as before are given by Eq. (3.5) and Y is used to identify a reference coordinate for the second axis.

An expansion for $\mathcal{A}(x,y)$ in the two-dimensional Chebyshev space W is

$$\mathcal{A}(x,y) \approx \sum_{k=0}^{M-1}\sum_{l=0}^{M-1} \hat{a}_{kl} T_k(X) T_l(Y), \quad (X,Y) \in I^2. \tag{3.18}$$

The coefficients $\hat{\boldsymbol{A}} = (\hat{a}_{kl})_{k,l=0}^{M-1} \in \mathbb{R}^{M\times M}$ are found by $L^2_\omega(I^2)$ orthogonal projection, or, more simply put, from a 2D forward Chebyshev transform. Multiplying Eq. (3.18) through by a test function $T_m(X)T_n(Y)$ and weights $\omega_X = (1-X^2)^{-1/2}$ and $\omega_Y = (1-Y^2)^{-1/2}$, integrating over the domain and using the orthogonality of Chebyshev polynomials leads to

$$\hat{a}_{kl} = \frac{4}{\pi^2 c_k c_l} \left(\left(\mathcal{A}(x,y), T_k(X)\right)_{\omega_X}, T_l(Y) \right)_{\omega_Y}, \forall\, (k,l) \in \mathcal{I}^2_M, \tag{3.19}$$

where $\mathcal{I}_M = \{0,1,\ldots,M-1\}$.

Keiner [8] shows that it is sufficient to use $M = 18$ for full double precision for the Chebyshev approximation of $\mathcal{A}(x,y)$ on any of the problems square submatrices. Hence the scalar products can also be computed without any loss of accuracy using numerical quadrature (instead of exact integration) on Chebyshev-Gauss points $\boldsymbol{X}^G = \boldsymbol{Y}^G = \{\cos\left((m+1/2)\pi/M\right)\}_{m=0}^{M-1}$, with constant quadrature weights $\{\pi/M\}_{m=0}^{M-1}$, and the definition $T_k(x) = \cos(k\cos^{-1}x)$, such that Eq. (3.19) becomes

$$\hat{a}_{kl} = \frac{4}{c_k c_l M^2} \sum_{m=0}^{M-1}\sum_{n=0}^{M-1} \mathcal{A}(x_m, y_n) \cos(k(m+1/2)\pi/M) \cos(l(n+1/2)\pi/M), \tag{3.20}$$

which is easily computed with a discrete cosine transform of type 2. Note that $x_m = i + h_\gamma(X^G_m + 1)$ and $y_n = j + h_\gamma(Y^G_n + 1)$ from Eq. (3.17).

We find the coefficients for all submatrices in $\boldsymbol{A}$ by running over all levels, blocks and submatrices, evaluating the matrix $(\mathcal{A}(x_m, y_n)) \in \mathbb{R}^{M\times M}$ and performing the discrete cosine transform for each square. This is the major cost of the initialization stage of the FMM. Since the initialization cost for each submatrix is the same (without using any optimizations), the total initialization cost will scale as $\mathcal{O}(N)$ since the number of submatrices grows asymptotically as $\mathcal{O}(N)$, see Eq. (3.10). In Sec. 4.3 we will discuss some significant optimizations for this initialization.

Note that the original multipole method of Alpert et al. [2] used nodal Lagrange polynomials $\{\ell_k(X)\}_{k=0}^{M-1}$ instead of Chebyshev,[2] and the approximation

$$\mathcal{A}(x,y) \approx \sum_{k=0}^{M-1}\sum_{l=0}^{M-1} \mathcal{A}(x_k, y_l) \ell_k(X) \ell_l(Y), \tag{3.21}$$

instead of Eq. (3.18). Hence, for the original method the initializtion cost is smaller, since it does not require the Chebyshev transform of the otherwise same matrix $(\mathcal{A}(x_m, y_n))_{m,n=0}^{M-1}$.

3.3. Evaluating the matrix vector product on one submatrix. For any given submatrix $\underline{\boldsymbol{A}} = \boldsymbol{A}(\gamma, b, r) \in \mathbb{R}^{2h_\gamma \times 2h_\gamma}$ the matrix vector product that we are interested in can be computed as

$$\underline{z}_m = \sum_{n=0}^{2h_\gamma - 1} \underline{a}_{mn} \underline{f}_n, \quad m = 0,1,\ldots,2h_\gamma - 1, \tag{3.22}$$

[2] To be perfectly consistent they also used the reference interval $[0,1]$ and not $[-1,1]$.

where $\underline{\boldsymbol{z}} = \boldsymbol{z}(\gamma, b, p)$, $\underline{\boldsymbol{f}} = \boldsymbol{f}(\gamma, b, q)$ and an underline notation is used in general to indicate a sub-vector/matrix.

If we do not take the sparsity of $\underline{\boldsymbol{A}}$ into consideration the flop count of (3.22) will be $8h_\gamma^2$. Note that we do not attempt to find an exact flop count and will only count the flops of the most deeply nested statements, using Table 1.1.2 of [6] for reference. For efficiency we now also split into odd and even parities:

$$\underline{z}_m^\sigma = \sum_{n=0}^{h_\gamma - 1} \underline{a}_{mn}^\sigma \underline{f}_n^\sigma, \quad \forall\, m \in \mathcal{I}_{h_\gamma} \text{ and } \sigma \in \{0, 1\}, \tag{3.23}$$

where $\mathcal{I}_{h_\gamma} = \{0, 1, \ldots, h_\gamma - 1\}$, $\underline{z}_m^\sigma = \underline{z}_{2m+\sigma}, \underline{f}_n^\sigma = \underline{f}_{2n+\sigma}$ and $\underline{a}_{mn}^\sigma = \underline{a}_{2m+\sigma,2n+\sigma}$. Since all zero items of $\boldsymbol{A}$ now have been eliminated, the cost has been reduced to $4h_\gamma^2$.

We now come to the actual multipole method and replace $\underline{a}_{mn}^\sigma$ by its Chebyshev approximation (3.18), such that the matrix vector product becomes

$$\underline{z}_m^\sigma = \sum_{n=0}^{h_\gamma - 1} \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} \hat{a}_{kl} T_k(X_m^{\sigma,\gamma}) T_l(Y_n^{\sigma,\gamma}) \underline{f}_n^\sigma, \quad m \in \mathcal{I}_{h_\gamma}, \sigma \in \{0, 1\}, \tag{3.24}$$

where the Vandermonde matrix $\boldsymbol{T}(\sigma, \gamma) = (T_k(X_m^{\sigma,\gamma}))_{m,k=0}^{h_\gamma - 1, M-1} \in \mathbb{R}^{h_\gamma \times M}$, where

$$X_m^{\sigma,\gamma} = Y_m^{\sigma,\gamma} = -1 + (2m + \sigma)/h_\gamma, \quad m \in \mathcal{I}_{h_\gamma}, \tag{3.25}$$

is the (uniform) discretization of the local reference coordinate. Note that the Chebyshev coefficients $\hat{a}_{kl}$ are the same for the odd or even computations, and the parity only affects the Vandermonde matrix. Also note that there is only one Vandermonde matrix for a given parity since the matrices $(T_k(X_m^{\sigma,\gamma}))$ and $(T_l(Y_n^{\sigma,\gamma}))$ in (3.24) are identical.

If we now rewrite (3.24) with matrix notation, $\hat{\boldsymbol{A}} = \hat{\boldsymbol{A}}(\gamma, b, r)$ and $\boldsymbol{T} = \boldsymbol{T}(\sigma, \gamma)$:

$$\underline{\boldsymbol{z}}^\sigma = \boldsymbol{T} \hat{\boldsymbol{A}} \boldsymbol{T}^T \underline{\boldsymbol{f}}^\sigma, \quad \forall\, \sigma \in \{0, 1\}, \tag{3.26}$$

we see that for given σ the cost is $2Mh_\gamma$ flops to do $\boldsymbol{w} = \boldsymbol{T}^T \underline{\boldsymbol{f}}^\sigma$, where $\boldsymbol{w} \in \mathbb{R}^M$ is a work array, $2M^2$ to do $\boldsymbol{c} = \hat{\boldsymbol{A}} \boldsymbol{w}$ for another work array $\boldsymbol{c} \in \mathbb{R}^M$, and $2h_\gamma M$ to do $\underline{\boldsymbol{z}}^\sigma = \boldsymbol{T}\boldsymbol{c}$, for a total cost of $4Mh_\gamma + 2M^2$. Adding the two odd and even parities together, the cost becomes $8Mh_\gamma + 4M^2$, so if $M << h_\gamma$ there can be a significant speedup over the $4h_\gamma^2$ cost of the direct computation.

3.4. Evaluating the matrix vector product for a block of submatrices. For any given *block* b of submatrices the matrix vector product that we are interested in can be computed by running over all 3 submatrices in the block and adding up the contributions of each, as shown in Alg. 3.1. The cost of this operation is less than 3 times $8Mh_\gamma + 4M^2$ (which is the cost for one submatrix isolated), because the vector block $\underline{\boldsymbol{f}}(q)$ is independent of p and $\underline{\boldsymbol{z}}(p)$ is independent of q. We get the flop count as detailed in Lemma 3.1.

LEMMA 3.1. *The significant flop count for a matrix vector product over a block is*

$$16h_\gamma M + 12M^2. \tag{3.27}$$

Proof. We can count the flops line by line in Alg. 3.1, taking only the most deeply nested computations into account. The first operation $\boldsymbol{T}^T \boldsymbol{f}(q)$ requires $8h_\gamma M$

Algorithm 3.1 Matrix vector product for a single block of submatrices for a given parity.

function MATVECBLOCK($\boldsymbol{f}, \boldsymbol{T}, \hat{\boldsymbol{A}}, \boldsymbol{z}$)
input $\boldsymbol{f}$: $\mathbb{R}^{2\times h}$ ▷ Legendre coefficients for given block and parity
input $\boldsymbol{T}$: $\mathbb{R}^{h\times M}$ ▷ Vandermonde matrix for given level and parity
input $\hat{\boldsymbol{A}}$: 3 sequence of $\mathbb{R}^{M\times M}$ ▷ Chebyshev matrices of coefficients
output $\boldsymbol{z}$: $\mathbb{R}^{2\times h}$ ▷ Chebyshev coefficients for given block and parity
  $\boldsymbol{c} \leftarrow 0$ ▷ Initialize work array $\boldsymbol{c} \in \mathbb{R}^{2\times M}$
  for $q \leftarrow 0, 1$ **do**
    $\boldsymbol{w} \leftarrow \boldsymbol{T}^T \boldsymbol{f}(q)$ ▷ Work array $\boldsymbol{w} \in \mathbb{R}^M$
    for $p \leftarrow 0, q$ **do**
      $r \leftarrow p + q(q+1)/2$ ▷ Eq. (3.3)
      $\boldsymbol{c}(p) \leftarrow \boldsymbol{c}(p) + \hat{\boldsymbol{A}}(r)\boldsymbol{w}$
    end for
  end for
  for $p \leftarrow 0, 1$ **do**
    $\boldsymbol{z}(p) \leftarrow \boldsymbol{z}(p) + \boldsymbol{T}\boldsymbol{c}(p)$
  end for
  return $\boldsymbol{z}$
end function

flops since the matrix vector product costs $2h_\gamma M$ and it is computed 2 times for each parity. The second step $\boldsymbol{c}(p) \leftarrow \boldsymbol{c}(p) + \hat{\boldsymbol{A}}(r(p,q))\boldsymbol{w}$ is a matrix vector product of cost $2M^2$, computed 3 times per parity. The third and final step $\boldsymbol{T}\boldsymbol{c}(p)$ costs $8h_\gamma M$ flops for the same reasons as the first step. Counting all three steps yields the number in Eq. (3.27). ∎

Algorithm 3.2 Procedure used to compute matrix vector product $\boldsymbol{z} = \boldsymbol{A}\boldsymbol{f}$.

for $\sigma \leftarrow 0, 1$ **do**
  for $\gamma \leftarrow 0, L-1$ **do**
    for $b \leftarrow 0, b_\gamma - 1$ **do**
      $\boldsymbol{z}^\sigma(\gamma, b) \leftarrow$ MATVECBLOC($\boldsymbol{f}^\sigma(\gamma, b)$, $\boldsymbol{T}(\sigma, \gamma)$, $\hat{\boldsymbol{A}}(\gamma, b)$, $\boldsymbol{z}^\sigma(\gamma, b)$)
    end for
  end for
end for

The complete matrix vector product can now be computed simply by running over all levels and blocks for odd and even parities of the matrix, as shown in the simplified Alg. 3.2. The total cost of the method can then be estimated as

$$\mu s N + \sum_{\gamma=0}^{L-1} b_\gamma \left(16 h_\gamma M + 12 M^2\right), \tag{3.28}$$

where $\mu s N$ represents the cost of using the direct method for the part of the matrix that is close to the diagonal and separated from the submatrices. The scaling factor $\mu = 3$ if the elements of the matrix $\boldsymbol{A}$ are precomputed. However, $\mu = 4.5$ if we simply use the vector $\boldsymbol{\lambda}$ as in Alg. 2.1. By inserting for b_γ and h_γ we can simplify

Eq. (3.28) further and obtain a total cost lower than

$$(\mu s + 4LM + \frac{6M^2}{s})N, \tag{3.29}$$

where the last number represents $4N_cM^2 = 12(\frac{N}{2s} - L - 2)M^2 < 6NM^2/s$, or the cost of applying all N_c submatrices $\hat{\boldsymbol{A}}(\gamma, b, r)$, each with a cost of $2M^2$ per parity. Since the number of levels $L \propto \log_2 N/s$, we get that the total cost of this very simple method scales as $\mathcal{O}(N \log_2 N)$.

4. A Faster Multipole Method. The Fast Multipole Method described in Sec. 3 is not optimal since the relatively expensive matrix vector products $\boldsymbol{T}^T \underline{\boldsymbol{f}}^\sigma(q)$ and $\boldsymbol{T}\boldsymbol{c}(p)$ in Alg. 3.1 are computed at each level. In the method described by Alpert and Rokhlin [2] this obstacle is overcome by their Eq. (4.8), which maps the Lagrange interpolating polynomials on one level to those on the next finer level. Below we will adapt a similar approach for the Chebyshev polynomials and this modification will make the total method scale as $\mathcal{O}(N)$. The complete faster algorithm is given in Alg. 4.1 and it is described in some more detail below.

4.1. The algorithm. The first objective of the faster method is to speed up the matrix vector product $\boldsymbol{w} = \boldsymbol{T}^T \underline{\boldsymbol{f}}^\sigma(q)$, which is computed 2 times per block on all levels. We thus start by predefining a multidimensional work array to hold this result for all levels: $\boldsymbol{w}(\gamma) \in \mathbb{R}^{b_\gamma \times 2 \times M} \, \forall \, \gamma \in \mathcal{I}_L$. For any level, block and parity the work vector $\boldsymbol{w}(\gamma, b, q) \in \mathbb{R}^M$ can be computed directly as the matrix-vector product

$$\boldsymbol{w}(\gamma, b, q) = \boldsymbol{T}^T(\sigma, \gamma)\boldsymbol{f}^\sigma(\gamma, b, q) \quad \in \mathbb{R}^M, \tag{4.1}$$

costing $2h_\gamma M$ operations per parity. As for the global vectors $\boldsymbol{f}$ and $\boldsymbol{z}$, we here look up the local part of the work vector using level γ, block b and local column index q. Since it is merely a work vector, we drop the parity superscript on $\boldsymbol{w}$.

The direct approach for computing $\boldsymbol{w}(\gamma)$ (step (1) in Alg. 4.1) will only be used for the highest level $\gamma = L-1$, where it can be set up efficiently as a tensor contraction

$$\boldsymbol{w}(L-1) = \boldsymbol{f}^\sigma(L-1)\boldsymbol{T}(\sigma, L-1) \quad \in \mathbb{R}^{b_{L-1} \times 2 \times M}, \tag{4.2}$$

between the two tensors $\boldsymbol{f}^\sigma(L-1) \in \mathbb{R}^{b_{L-1} \times 2 \times s}$ and $\boldsymbol{T}(\sigma, L-1) \in \mathbb{R}^{s \times M}$. The cost is approximately $4b_{L-1}Ms = (N - 4s)M < MN$ flops per parity.

For all lower levels it is shown in some detail in Appendix A.1 that the matrix vector product (4.1) can be computed as (step (2) in Alg. 3.2)

$$\boldsymbol{w}(\gamma, b, q) = \sum_{l \in \{0,1\}} \boldsymbol{B}(l)\boldsymbol{w}(\gamma + 1, 2b + q + 1, l), \tag{4.3}$$

where $\boldsymbol{B}(l) = (b_{kn}^{(l)})_{k,n=0}^{M-1}$ are the lower triangular matrices that map the basis function T_k used on the half-interval $I(l)$ of level γ (see illustration for matrix $A(\gamma, b, 0)$ in Fig. 2) to the full-interval Chebyshev polynomials used for the matrices directly below on level $\gamma + 1$. That is

$$T_k((Y + 2l - 1)/2) = \sum_{j=0}^{k} b_{kj}^{(l)} T_j(Y), \quad Y \in I, \, l \in \{0, 1\}, \tag{4.4}$$

where $(Y+2l-1)/2 \in I(l) = [l-1, l]$. The two lower triangular matrices $\boldsymbol{B}(l) \in \mathbb{R}^{M \times M}$ can be computed exactly using the binomial theorem and simple basis changes, see

appendix A.3. Equation (4.4) is the modal version of Eq. (4.9) used in [2]. However, the corresponding matrices (i.e, $(u_r(t_i/2))_{r,i=0}^{M-1}$ in Eq. (4.9) of [2]) required by the Lagrange interpolating polynomials are both dense.

Equation (4.3) consists of the sum of two matrix vector products, where the matrices $\boldsymbol{B}(l) \in \mathbb{R}^{M\times M}$ are lower triangular. The cost would thus normally be $2M^2$ flops for each parity and nearly twice as fast as the dense approach used by Alpert and Rokhlin [2]. However, we can also take advantage of the fact that the two matrices $\boldsymbol{B}(0)$ and $\boldsymbol{B}(1)$ are very similar (see Eq. (A.7) and App. A.3, $\boldsymbol{B}(0)$ and $\boldsymbol{B}(1)$ differ only in the sign of the odd diagonals), and the cost may then be reduced even further to one single lower triangular matrix vector product (M^2 flops) plus a small additional cost of $2M$ for preparing two vectors of length M. The flop count is thus nearly 4 times less than a dense approach. A faster algorithm is described in detail in Alg. A.1.

Step (3) of Alg. 4.1 is simply the application of the precomputed coefficient matrices $\hat{\boldsymbol{A}}(\gamma, b, r)$ to the computed work vector $\boldsymbol{w}(\gamma, b, q)$ on each submatrix. The cost is here the same $4N_cM^2 < 6NM^2/s$ as computed in Eq. (3.29).

Step (4) of Alg. 4.1 is the second matrix vector product from Alg. 3.1 ($\boldsymbol{z} = \boldsymbol{T}\boldsymbol{c}(p)$) that needs to be handled more efficiently in order to obtain a $\mathcal{O}(N)$ method. Since this is basically the transpose of Step (2) we can also here make use of the same matrices $\boldsymbol{B}(l)$. However, the computed result will now have to be transported in the reverse direction, from the lowest level 0 and up to the highest level $L-1$. Since $\boldsymbol{B}(l)$ is triangular the cost for the two matrix vector products in the innermost loop of step (4) will be $2M^2$ flops per parity. This can be further improved to $\approx M^2$ flops by taking advantage of the structure of $\boldsymbol{B}(0)$ and $\boldsymbol{B}(1)$, similar to Step (2).

Having transported the intermediate array $\boldsymbol{c}$ up to the highest level, $L-1$, a tensor contraction (Step (5) in Alg. 4.1) of the same cost as Step (1) concludes the FMM part of the method

$$\boldsymbol{z}^{\sigma}(L-1) = \boldsymbol{c}(L-1)\boldsymbol{T}^T(\sigma, L-1). \tag{4.5}$$

The complete and faster $\mathcal{O}(N)$ Legendre to Chebyshev (L2C) algorithm presented in Alg. 4.1 is basically 5 different matrix vector products (or tensor contractions) and simple enough that it can be implemented in less than 100 lines of high-level Python code. The algorithm for the reverse direction (C2L) simply requires that the diagonal matrix $\boldsymbol{C}$ is applied to the input vector first (see Eq. (1.5)), and otherwise the only difference is that $\mathcal{L}(x, y)$ (see Eq. (4.10) and Eq. (2.21) of [2]) is used instead of $\mathcal{A}(x, y)$ for the Chebyshev approximations in Eqs. (3.19) and (3.20).

4.2. Total cost of faster method.

LEMMA 4.1. *Retaining only the most deeply nested loops, the total flop count of the faster method described in Alg. 4.1 can be estimated as*

$$\left(\mu s + 4M + \frac{(6+2)M^2}{s}\right) N. \tag{4.6}$$

Proof. The algorithm is divided into 7 steps, as shown in Alg. 4.1. There is no difference in cost from (3.29) for step (6): $\mu s N$ or step (3): $6NM^2/s$. For steps (1) and (5) the tensor contractions each cost less than MN flops (see Eq. (4.2)) for each parity and thus a total of $4MN$. The two matrix vector products in the innermost loop of both steps (2) and (4) cost for each parity $2M^2$ plus a small and neglected cost for setting up vectors of length M. These products are applied for all but one block on

Algorithm 4.1 Complete faster algorithm for the Legendre to Chebyshev transform (L2C) $\boldsymbol{z} = \boldsymbol{C}^{-1}\boldsymbol{A}\boldsymbol{f}$.

function LEG2CHEB($L, \boldsymbol{f}, \boldsymbol{T}, \hat{\boldsymbol{A}}, \boldsymbol{B}, \boldsymbol{C}$)
input L : $\mathbb{N}$ ▷ Number of levels
input $\boldsymbol{f}$: $\mathbb{R}^N$ ▷ Legendre coefficients
input $\boldsymbol{T}$: $\mathbb{R}^{2\times s\times M}$ ▷ Vandermonde matrices, $s = h_{L-1}$
input $\hat{\boldsymbol{A}}$: N_c sequence of $\mathbb{R}^{M\times M}$ ▷ Chebyshev matrices of coefficients
input $\boldsymbol{B}$: $\mathbb{R}^{2\times M\times M}$ ▷ Lower triangular matrices
input $\boldsymbol{C}$: matrix $\in \mathbb{R}^{N\times N}$ ▷ diagonal matrix, see Eq. (1.4)
output $\boldsymbol{z}$: $\mathbb{R}^N$ ▷ Chebyshev coefficients
  for $\sigma \leftarrow 0, 1$ **do**
    for $\gamma \leftarrow 0, L-1$ **do** ▷ Initialize work arrays
      $\boldsymbol{c}(\gamma) \leftarrow 0$ ▷ Work array $\mathbb{R}^{b_\gamma\times 2\times M}$
      $\boldsymbol{w}(\gamma) \leftarrow 0$ ▷ Work array $\mathbb{R}^{b_\gamma\times 2\times M}$
    end for
    $\boldsymbol{w}(L-1) \leftarrow \boldsymbol{f}^\sigma(L-1)\boldsymbol{T}(\sigma)$ ▷ See Eq. (4.2). Step (1)
    for $\gamma \leftarrow L-1, 1$ **do** ▷ See Eq. (4.3). Step (2)
      for $b \leftarrow 1, b_\gamma - 1$ **do**
        $b_0, q_0 \leftarrow$ **divmod**$(b-1, 2)$
        $\boldsymbol{w}(\gamma-1, b_0, q_0) = \boldsymbol{B}(0)\boldsymbol{w}(\gamma, b, 0) + \boldsymbol{B}(1)\boldsymbol{w}(\gamma, b, 1)$
      end for
    end for
    for $\gamma \leftarrow 0, L-1$ **do** ▷ Step (3)
      for $b \leftarrow 0, b_\gamma - 1$ **do**
        for $q \leftarrow 0, 1$ **do**
          for $p \leftarrow 0, q$ **do**
            $r \leftarrow p + q(q+1)/2$ ▷ See Eq. (3.3)
            $\boldsymbol{c}(\gamma, b, p) \leftarrow \boldsymbol{c}(\gamma, b, p) + \hat{\boldsymbol{A}}(\gamma, b, r)\boldsymbol{w}(\gamma, b, q)$
          end for
        end for
      end for
    end for
    for $\gamma \leftarrow 0, L-2$ **do** ▷ Step (4)
      for $b \leftarrow 0, b_{\gamma+1} - 2$ **do**
        $b_0, q_0 \leftarrow$ **divmod**$(b, 2)$
        for $p \leftarrow 0, 1$ **do**
          $\boldsymbol{c}(\gamma+1, b, p) \leftarrow \boldsymbol{c}(\gamma+1, b, p) + \boldsymbol{B}^T(p)\boldsymbol{c}(\gamma, b_0, q_0)$
        end for
      end for
    end for
    $\boldsymbol{z}^\sigma(L-1) \leftarrow \boldsymbol{c}(L-1)\boldsymbol{T}^T(\sigma)$ ▷ See Eq. (4.5). Step (5)
  end for
  Apply direct method to remaining part ▷ Step (6)
  $\boldsymbol{z} \leftarrow \boldsymbol{C}^{-1}\boldsymbol{z}$ ▷ Step (7)
return $\boldsymbol{z}$
end function

all levels, which counts to $\sum_{\gamma=1}^{L-1}(b_\gamma - 1) = 2(2^L - L - 1) < \frac{N}{2s}$ blocks (see Eqs. (3.10) and (3.7)). Steps (2) and (4) thus costs $\approx 4M^2 \cdot N/2s = 2M^2N/s$ flops. Step (7) is simply N multiplications and thus neglected along with similar small operations. ∎

It is noteworthy that steps (2) and (4) now together cost approximately $2M^2N/s$ flops and much less than step (3) (i.e., $6M^2N/s$). For the nodal Lagrange approach steps (2) and (4) require approximately $8M^2N/s$ flops.

Assuming that computational speed is simply proportional to the number of flops, we can compute the optimal size of the smallest submatrices from Eq. (4.6). Setting $\mu = 3$ and $M = 18$, we find that the optimal s is 29, and for a power of 2 input this should be rounded up to $s = 32$, which is the number used by Alpert et al. [2]. For these parameters the modal L2C requires approximately $250N$ flops for execution, which is nearly 20 % less than the nodal L2C ($310N$). We note that a DCT type 2 costs $\approx 2N \log_2 N$ flops (see [14]) and the modal FMM is thus using $125/\log_2 N$ times more flops than a DCT.

However, computational speed is not straightforward and computer, code and compiler settings make the problem of choosing optimal s more complicated. For our C code [11] and the computers used in Sec 5, it is only optimal to use precomputed matrix entries and $s = 32$ without compiler optimization. For more aggressive and vectorized compiler flags (e.g., -march=native, -Ofast) it is actually more efficient not to precompute the matrix entries and also to use $s = 64$ instead of $s = 32$. The compiler optimization mostly improves the direct part (step (6)) of the algorithm and this indicates that memory access is a bottleneck for the precomputed approach, as discussed in Sec. 2.[3]

Finally, we note that the memory requirement for the method described in Alg. 4.1 is merely $10N$ numbers of the chosen double precision, if we choose not to precompute $\boldsymbol{A}$ for the direct part. This is the requirement for storing N_c matrices $\hat{\boldsymbol{A}}$, each of shape $\mathbb{R}^{M\times M}$, and two vectors of shape $\mathbb{R}^N$ (a work array and $\boldsymbol{\lambda}$). Everything else is negligible in terms of memory (assuming $N \gg s$). If we choose to precompute entries of $\boldsymbol{A}$ for the direct part, there is an additional and significant memory requirement of $1.5sN$ numbers.

4.3. A faster initialization. The major cost for the initialization of the fast multipole method is to run over all submatrices, evaluate (see Eq. (3.20))

$$\mathcal{A}(x_m, y_n) = \Lambda\left(\frac{y_n - x_m}{2}\right)\Lambda\left(\frac{y_n + x_m}{2}\right), \quad 0 \le m, n < M, \tag{4.7}$$

and apply a two-dimensional DCT of type 2 to each matrix $(\mathcal{A}(x_m, y_n))_{m,n=0}^{M-1}$. We note that the matrix $(\mathcal{A}(x_m, y_n))_{m,n=0}^{M-1}$ is the Hadamard product of two matrices, that for simplicity will be referred to in this section as $\Lambda^- = (\Lambda((y_n - x_m)/2))_{m,n=0}^{M-1}$ and $\Lambda^+ = (\Lambda((y_n + x_m)/2))_{m,n=0}^{M-1}$.

For the initialization it is very important that $\Lambda(z)$ is computed efficiently. Alpert et al. [2] describe how asymptotically $\Lambda(z) \sim \frac{1}{\sqrt{z}}$ as $z \longrightarrow \infty$ such that $\Lambda(z)\sqrt{z}$ can be well approximated by a polynomial series in $1/z$ of length 6. An even better approach along the same lines is suggested by Bogaert et al. [3] and Slevinsky [17],

[3] When examining the optimized assembly code produced for both the precomputed and non-precomputed versions of Alg. 2.1, there are only a few minor differences for the inner loop. In the precomputed version, more data is transferred into SIMD (see [1]) registers (via 'vmovsd') than for the non-precomputed version, where there instead are more multiplications (via 'vmulsd') with data already residing within the registers.

which is to adapt a Taylor series in $1/z$ for the function $\tau(z) = \sqrt{z}\frac{\Gamma(z+1/4)}{\Gamma(z+3/4)}$ and then compute $\Lambda(z) = \tau(z+1/4)/\sqrt{z+1/4}$. This latter approach is better because the Taylor expansion is more quickly decaying, with only even terms remaining:

$$\tau(z) \approx 1 - \frac{1}{2^6 z^2} + \frac{21}{2^{13} z^4} - \frac{671}{2^{19} z^6} + \frac{180,323}{2^{27} z^8} + \mathcal{O}\left(\frac{1}{z^{10}}\right). \tag{4.8}$$

Another advantage is that all but the last of these rational coefficients can be represented exactly in double precision since the denominator of the coefficient is small enough and a power of 2. Using Eq. (4.8) we find that $\Lambda(z)$ can be computed with relative error to within machine precision (i.e., $\approx 2.22 \times 10^{-16}$) for all $z > 19.88$. However, as $z \longrightarrow \infty$ more terms on the right hand side can be discarded and it is easy to verify numerically that with only $4, 3$ or 2 terms, the expression is still accurate to machine precision for $z > 40,134$ or 1844, respectively.[4] Hence, the cost for evaluating $\mathcal{A}(x_m, y_n)$ will be smaller for large N. The smallest relevant (x, y) for the submatrices are found on the highest level $L-1$ closest to the main diagonal where $(y-x)/2 \geq s$. Hence, with $s = 64$, we only need the first 4 terms of Eq. (4.8) on the highest level $L-1$. For levels $L-2, L-3, L-4$ and $L-5$ it is sufficient to use 3 terms and for all levels $< L-5$ merely 2 terms of Eq. (4.8) are required. We can estimate that $\Lambda(z)$ costs 20 flops for the highest level (including ≈ 10 for a square root), and then remove 3 flops per removed term of Eq. (4.8) for the higher levels.

There are still significant enhancements that can be made for more efficient evaluation of the matrix $(\mathcal{A}(x_m, y_m))$, by taking advantage of certain symmetries. First of all, the matrix Λ^- is persymmetric, whereas Λ^+ is symmetric. Hence, symmetry dictates that only $(M^2+M)/2$ items of each matrix need to be evaluated, which nearly halves the flop count of evaluation. Second, since Λ^- is persymmetric and only depends on the distance to the main diagonal of the matrix $\boldsymbol{A}$ (see Eq. (1.8)), it is sufficient to evaluate Λ^- for the first two submatrices of the first block of any given level (i.e., for the submatrices $\boldsymbol{A}(\gamma, 0, 0)$ and $\boldsymbol{A}(\gamma, 0, 1)$, see Eq. (3.18)). All remaining submatrices on that level will then be able to reuse one of these two already evaluated matrices. Asymptotically this eliminates the cost for evaluating Λ^-, as all it takes now is $L(M^2+M)$ evaluations, with $L \sim \log_2 N$, see Eq. (3.9). Since computing Λ^- is half the cost of evaluating $(A(x_m, y_n))$ the flop count is thus reduced by nearly another factor 2. With these optimizations the flop count for the evaluation drops nearly by a factor 4 from the naïve evaluation of $\mathcal{A}(x_m, y_n)$ at all points. Assuming $\Lambda(z)$ costs 20 flops, then, on average, the cost will be $10M^2$ ($40M^2$ for naïve, divided by 4) flops for each submatrix.

For the methods described in Secs. 3 and 4 we also need to take the DCT of all the matrices $(\mathcal{A}(x_m, y_n))$. The cost of this DCT will depend on the implementation, but it should be close to $2M \log_2 M$, see [14]. We have implemented the DCT of size $M = 18$ using a mixed radix 2 and 3, since $18 = 2 \cdot 3^2$, at a cost close to $2M \log_2 M$ flops. Hence, the 2D DCT on a submatrix, where the 1D DCT is applied to $2M$ vectors of length M, can be computed in approximately $4M^2 \log_2 M$ arithmetic operations.

For the direct part of the transform (step 6) we need to precompute the vector $\boldsymbol{\lambda} = \{\Lambda(i)\}_{i=0}^{N-1}$. A direct application of Eq. (4.8) at 20 flops per evaluation will here lead to an initialization cost of $\approx 20N$ flops. Some of these could be stored in a lookuptable, but for integer arguments to $\Lambda(z)$ a more elegant solution is to use

[4] Here rounding up to nearest integer.

$\Lambda(0) = \Gamma(1/2) = \sqrt{\pi}$ and then the recursion

$$\Lambda(i) = \Lambda(i-1)\frac{i-1/2}{i}, \quad \forall\, i = 1, 2, \ldots, N-1, \tag{4.9}$$

which is stable since $(i - 1/2)/i$ remains close to, and is always smaller than, 1. The absolute error of computing Eq. (4.9) is found to be less than machine precision for all $i < 2^{23}$. However, the error of each iteration is found to be 2 ulps (unit in the last place) or less, and these may accumulate. In order to achieve a relative error for all $i < 2^{23}$ that is less than 10^{-15} and an absolute error less than 5 ulps, we have found it necessary to compute every 8'th $\Lambda(i)$ from Eq. (4.8) and the remaining 7 out of 8 from (4.9). The cost for computing $\boldsymbol{\lambda}$ will then be approximately $\frac{7}{8}2N + \frac{1}{8}20N \approx 4N$ flops. If all items of the matrix $\boldsymbol{A}$ that are used in the direct part are precomputed, then the initialization cost will be $4N + 1.5sN$ flops, since in addition to computing $\boldsymbol{\lambda}$, we would also need to multiply two $\Lambda(i)$'s together for each of the $1.5sN$ nonzero items. However, since it has been found faster to not use precomputation, and since the memory requirement is many times larger and the initialization takes much longer, we will only consider the non-precomputed version of the algorithm in what follows.

In summary, the cost for the Legendre to Chebyshev (L2C) initialization may be estimated as a small fixed cost plus $N_c(4M^2 \log_2 M + 10M^2) + 4N$ flops asymptotically. Inserting for N_c we obtain $\approx N(6M^2 \log_2 M + 15M^2)/s + 4N$ flops. For the nodal Lagrange method the DCT, and thus the term $6M^2 \log_2 M$, can be dropped.

Finally, we note that the transform in the reverse direction, Chebyshev to Legendre (C2L), can be computed without any further evaluations of the $\Lambda(z)$ function. This follows since the transform function for the reverse direction, $\mathcal{L}(x, y)$, can be written such as to reuse Λ^- and Λ^+ for its nonzero items:

$$\mathcal{L}(x, y) = \frac{2(x + 1/2)y}{(x + y)(x + y + 1)(x - y + 1)} \frac{\Lambda\left((y - x)/2\right)}{\Lambda\left((x + y)/2\right)}. \tag{4.10}$$

Equation (4.10) is a simple reformulation of Eq. (2.21) of [2], using Eq. (4.9) and $\Lambda(i - 1/2) = (i\Lambda(i))^{-1}$.

5. Results. All numerical results are computed on a MacBook Pro (2.3 GHz 8-Core Intel Core i9) and/or on a single core of the SAGA supercomputer (Intel Xeon-Gold 6138 2.0 GHz). The algorithms have been implemented in both C and Python, and the code is available in the public repository https://github.com/spectralDNS/Legendre-to-Chebyshev. The Python code is not very efficient and mainly intended to illustrate the simplicity of the algorithm. The C-code is more optimized for speed, with the most heavy duty operations outsourced to OpenBLAS [21]. The triangular matrix-vector products in steps (2) and (4) and the direct step (6) are the only parts that are implemented in detail by hand. All computations except $N = 256$ use constant $M = 18$ and $s = 64$ since this has been found optimal in terms of execution speed. For $N = 256$ we use $s = 32$. For comparison, Alpert and Rokhlin [2] used $M = 18$ and $s = 32$[5]. The direct part (step (6)) of Alg. 4.1 is performed without precomputing matrix entries.

To evaluate the numerical accuracy we use the relative maximum norm

$$E_\infty = \frac{||\boldsymbol{z} - \boldsymbol{z}^*||_\infty}{||\boldsymbol{z}^*||_\infty}, \tag{5.1}$$

[5]There is a factor 2 difference between our definition of s and the parameter s used by [2]. This is because we need the submatrix-sizes to be even and thus define the smallest submatrices to have size $2s$ instead of s.

Table 1: Maximum error E_∞ of Legendre-to-Chebyshev (L2C) and Chebyshev-to-Legendre (C2L) transforms for the current modal (m) FMM and the nodal (n) FMM of [2]. The input array is drawn from random numbers in the range $[-1, 1]$ and the error is computed with (5.1) using a direct multiprecision solver for $\boldsymbol{z}^*$. The half integer number in parenthesis is the error in $||\boldsymbol{z} - \boldsymbol{z}^*||_\infty$ as the number of ulps of $||\boldsymbol{z}^*||_\infty$.

N	L2C (m)		L2C (n)		C2L (m)		C2L (n)	
	E_∞	(ulp)	E_∞	(ulp)	E_∞	(ulp)	E_∞	(ulp)
512	9.61e-16	(6.0)	9.61e-16	(6.0)	7.28e-16	(6.0)	7.28e-16	(6.0)
1024	9.32e-16	(6.0)	9.32e-16	(6.0)	1.31e-15	(6.5)	1.31e-15	(6.5)
2048	7.47e-16	(5.0)	5.98e-16	(4.0)	1.17e-15	(10.0)	1.17e-15	(10.0)
4096	9.02e-16	(6.0)	9.02e-16	(6.0)	2.37e-15	(12.5)	2.37e-15	(12.5)
8192	9.03e-16	(6.0)	9.03e-16	(6.0)	1.47e-15	(12.0)	1.47e-15	(12.0)
16384	9.01e-16	(6.0)	9.01e-16	(6.0)	1.43e-15	(8.5)	1.43e-15	(8.5)
32768	6.83e-16	(4.5)	6.07e-16	(4.0)	1.64e-15	(14.0)	1.64e-15	(14.0)
65536	6.76e-16	(4.5)	6.01e-16	(4.0)	1.75e-15	(10.5)	1.75e-15	(10.5)

where $\boldsymbol{z}$ is the computed result and $\boldsymbol{z}^*$ is the 'exact' result computed with a direct multiprecision C++ solver, accurate to approximately 32 digits. We will also evaluate two consecutive forward and backward transforms, in which case the exact result will be the input data.

We use random coefficients that decay as N^{-r}, where $r \geq 0$, see, e.g., [16, 7, 20]. The random coefficients are drawn uniformly from the interval $[-1, 1]$ using the standard C function **rand** and for repeatability we use the constant seed **srand(1)**. At first we consider only the accuracy of the Legendre-to-Chebyshev (L2C) and Chebyshev-to-Legendre (C2L) methods for uniform coefficients with $r = 0$. Since the multiprecision implementation uses a direct method we limit the test for coefficient vectors of size up to $N = 2^{16}$. Table 1 shows that the algorithm is accurate to all but one digit for both directions and there is very little error growth with size. Also, there is hardly any difference in the E_∞ norm between the modal approach described in this paper and the original nodal approach of [2]. The apparent coincidence that most E_∞ numbers are the same for both nodal and modal approaches in Table 1 is attributed to the fact that the error $||\boldsymbol{z} - \boldsymbol{z}^*||_\infty$ is merely a small integer or half-integer times one ulp of $||\boldsymbol{z}^*||_\infty$ (the value of the ulp depends on $||\boldsymbol{z}^*||_\infty$). The error in $||\boldsymbol{z} - \boldsymbol{z}^*||_\infty$ in ulps of $||\boldsymbol{z}^*||_\infty$ is given in parenthesis in Table 1.

In Fig. 3 we show the error of computing one L2C followed by a C2L, which should return the input array. For this test we use both uniform random coefficients (r=0) and coefficients decaying with r=1/2. The latter is more realistic since Legendre (or Chebyshev) coefficients usually decay in applications where the solution is smooth. We use only the modal FMM since the difference in accuracy from the nodal is minimal. From Fig. 3 we see that the error for the decaying input array is close to machine precision (one ulp for this plot equals 1.11×10^{-16}) for N all the way up to 10 million, whereas the uniform input leads to slightly decreased accuracy that becomes notable for $N > 100,000$. Similar bounded $\mathcal{O}(1)$ accuracy was also obtained by the Toeplitz-Hadamard (TH) method of Townsend et al. [20] for decay rates r $\geq 1/2$, whereas they observed $\mathcal{O}(N^{0.5} \log N)$ error growth for r=0. However, they only presented results for $N < 1000$. Results for N up to 10^7 were presented for the divide and conquer model by Olver et al. [13], showing a $\mathcal{O}(1)$ error for r=1. However, their simulation with

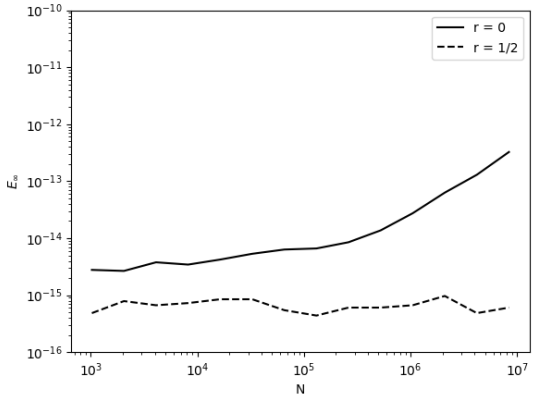
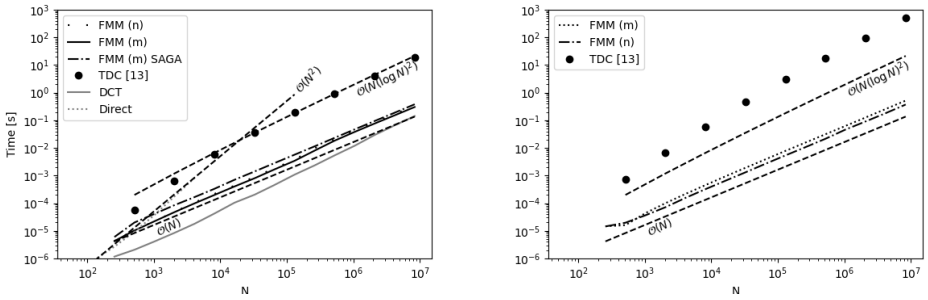


Fig. 3: The normalized maximum error for a consecutive L2C and C2L transforms. The black solid curve is using uniform random input data (r=0) and the dashed curve is using decaying, random Legendre coefficients scaled with r=1/2.

$N = 8388608$ required 10^{11} bytes of memory, see Table 4.3 of [13]. By comparison, the current method requires less than 10^9 bytes ($10N$ double precision numbers) for the same problem size. Finally, we note that the error growth was considerably larger both in Hale et al. [7] and Shen et al. [16], whereas Keiner [8], who used the original FMM method, could also show $\mathcal{O}(1)$ error growth for r=0 and $N \leq 10^5$ (see Table 6.2 of [8]).

The speed (wall clock time) of the current modal L2C[6] transform is shown in Figure 4 (a) with a black solid curve for the MacBook and a dash-dotted curve for the SAGA supercomputer. The large black dots show the data reported in Table 4.3 of [13] for the triangular-banded divide and conquer (TDC) method. The gray solid curve shows the execution time of a DCT type 2 computed with precompiled (conda-forge, using –enable-sse2 and –enable-avx) FFTW [5] on the MacBook, using the FFTW_MEASURE planning flag. The $\mathcal{O}(N)$ scaling is obvious for the current FMM method on the SAGA computer, but we struggle to maintain perfect scaling for $N > 10^5$ on the Mac. This slight degradation has been found to be related to compiler optimizations (-march=native, -Ofast), because a less aggressive approach (-O1) can be shown to maintain a linear scaling throughout, albeit at half the speed for small N. Still, it is noteworthy that a transform the size of $N = 10^7$ can be obtained in approximately 0.4 seconds with the current FMM, which is approximately three times slower than a type 2 DCT computed with FFTW. For large N the speed of FMM can bee seen to be nearly 2 orders of magnitude faster than the execution times reported for the TDC method [13]. A large part of this factor may be due to implementation, but the results of TDC are also reported to be faster than the comparative Toeplitz-Hankel method described in [20] (see Fig 5 (right)), which again are reported to be faster than the method of Hale et al. [7]. The speed of the nodal FMM using Lagrange polynomials is shown as a loosely dotted curve, and it falls very close to the results of the modal FMM, being on average $\approx 10\%$ slower (the nodal method is closer to the modal when using $s = 64$ instead of $s = 32$). The slower execution is

[6] The speed of the reverse direction C2L is basically identical, so only one direction is shown.



(a) Execution times. (b) Planning times.

Fig. 4: In (a) the black solid and dash-dotted curves show the execution times in seconds for a Legendre to Chebyshev transform of the current FMM (m) on the MacBook and SAGA computers, respectively. The loosely dotted curve shows the execution time (Mac) for the nodal FMM (n). The large black dots show the execution times of the triangular-banded divide and conquer (TDC) method from Table 4.3 of [13]. The execution time (Mac) for a type 2 DCT (FFTW) is shown in gray. In (b) the planning time (Mac) for the FMM (m) is shown as a black dotted line, for FMM (n) in a dash-dotted line and for TDC [13] in large dots. Three dashed lines are shown with increasing slope to illustrate the asymptotic behaviour for $\mathcal{O}(N), \mathcal{O}(N(\log N)^2)$ and $\mathcal{O}(N^2)$. The asymptotic curves are identical in (a) and (b).

only due to the transport of intermediate results through steps (2) and (4) in Alg. 4.1. Otherwise, the execution makes use of exactly the same code, only with different matrices and coefficients. A direct comparison with execution times of Shen et al. [16] is not possible, because they only report the flop count of their execution. However, comparing to Fig 3 in [16], the current minimal flop count of $\approx 250N$ is at least a factor 2 lower.

In Fig. 4 (b) we show the time spent on planning the FMM transforms, both for the current method (black dotted line) and for the Lagrange FMM method (dash-dotted line) using the optimizations described in Sec 4.3 without precomputation of matrix entries. The TDC from Table 4.3 of [13] is shown as large black dots. We observe the claimed $\mathcal{O}(N)$ scaling, and a modal FMM planning that takes approximately two times longer than execution. The optimized planning stage for the nodal FMM is very nearly as fast as the execution. We note that the nodal FMM [2] is faster to plan since it does not compute the DCT on its data. Otherwise the planning is very similar. Also note that using a larger value for s, like $s = 128$, the planning will become faster, whereas the execution becomes slower. This is because there are then fewer submatrices to initialize, whereas the direct part of the execution becomes more expensive.

6. Conclusions. In this paper we have described a modal version of the originally nodal Fast Multipole Method (FMM) of Alpert and Rokhlin [2], used for transforming between Legendre and Chebyshev coefficients. The modal approach allows us to spread intermediate results faster through different levels of hierarchical matrices, allowing us to speed up the original method with approximately $10-20$ %, depending

on parameter settings. For a power of 2 double precision input array, we can perform the modal transform in approximately $250N$ flops, compared to $310N$ for the nodal approach. However, the modal approach is slower to plan, due to an additional discrete cosine transform required by the initialization. The accuracy is the same for both nodal and modal versions, and we can transform millions of slightly decaying coefficients at nearly machine precision.

We have described in detail an algorithm that is easy to implement, relying mostly on a series of dense matrix-vector products that can be executed efficiently through BLAS. A C implementation is shown to be able to transform $N = 10^6$ double precision coefficients in approximately 40 milliseconds on a MacBook Pro laptop. This is approximately three times slower than a well-planned discrete cosine transform of type 2 from FFTW [5] on the same machine. For large N the planning of the transform, which needs to be done just once for given N, is approximately 2 times slower than the execution, and the memory use for a double precision transform is no more than $10N$ numbers. We have also described optimizations for the nodal FMM, which lead to at least 4 times faster initialization than originally described.

The method described in this paper is easily adapted to similar transforms between any bases of Gegenbauer polynomials, and not only Legendre-Chebyshev. For general Jacobi polynomials, that are not necessarily odd or even functions, the method can be extended by using a full upper triangular connection matrix.

7. Code availability. The code used to implement the method described in this paper is open source and available from the repository https://github.com/spectralDNS/Legendre-to-Chebyshev. Here both the original nodal FMM and the new modal FMM are implemented. The method is already in use in the spectral Galerkin framework Shenfun [12].

8. Acknowledgements. Some computations were performed on resources provided by Sigma2 - the National Infrastructure for High Performance Computing and Data Storage in Norway.

REFERENCES

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2023, https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.
[2] B. K. Alpert and V. Rokhlin, *A Fast Algorithm for the Evaluation of Legendre Expansions*, SIAM Journal on Scientific and Statistical Computing, 12 (1991), pp. 158–179, https://doi.org/10.1137/0912009.
[3] Bogaert, I. and Michiels, B. and Fostier, J., $\mathcal{O}(1)$ *Computation of Legendre Polynomials and Gauss–Legendre Nodes and Weights for Parallel Computing*, SIAM Journal on Scientific Computing, 34 (2012), pp. C83–C101, https://doi.org/10.1137/110855442.
[4] T. A. Driscoll, N. Hale, and L. N. Trefethen, *Chebfun guide*, 2014.
[5] M. Frigo and S. G. Johnson, *FFTW: An Adaptive Software Architecture for the FFT*, Proceedings of the IEEE, 93 (2005), pp. 216–231, https://doi.org/10.1109/JPROC.2004.840301.
[6] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 4th ed., 2013.
[7] N. Hale and A. Townsend, *A Fast, Simple, and Stable Chebyshev-Legendre Transform Using an Asymptotic Formula*, SIAM J. Sci. Comput., 36 (2014).
[8] J. Keiner, *Computing with Expansions in Gegenbauer Polynomials*, SIAM Journal on Scientific Computing, 31 (2009), pp. 2151–2171, https://doi.org/10.1137/070703065.
[9] J. Keiner, *Fast Polynomial Transforms*, Logos Verlag Berlin GmbH, 2011.
[10] R. J. Mathar, *Chebyshev series expansion of inverse polynomials*, Journal of Computational and Applied Mathematics, 196 (2006), pp. 596–607, https://doi.org/https:

//doi.org/10.1016/j.cam.2005.10.013, https://www.sciencedirect.com/science/article/pii/S0377042705006230.
[11] M. Mortensen, *Legendre-to-Chebyshev: Routines for Legendre to Chebyshev (and inverse) transforms*, https://github.com/spectralDNS/Legendre-to-Chebyshev.
[12] M. Mortensen, *Shenfun: High performance spectral Galerkin computing platform*, Journal of Open Source Software, 3 (2018), p. 1071, https://doi.org/10.21105/joss.01071.
[13] S. Olver, R. M. Slevinsky, and A. Townsend, *Fast algorithms using orthogonal polynomials*, Acta Numerica, 29 (2020), p. 573–699, https://doi.org/10.1017/S0962492920000045.
[14] X. Shao and S. G. Johnson, *Type-II/III DCT/DST algorithms with reduced number of arithmetic operations*, Signal Processing, 88 (2008), pp. 1553–1564, https://doi.org/10.1016/j.sigpro.2008.01.004.
[15] J. Shen, *Efficient chebyshev-legendre galerkin methods for elliptic problems*, in ICOSAHOM, 1996.
[16] J. Shen, Y. Wang, and J. Xia, *Fast Structured Jacobi-Jacobi Transforms*, Math. Comp., 88 (2019), pp. 1743–1772.
[17] R. M. Slevinsky, *Fast and backward stable transforms between spherical harmonic expansions and bivariate Fourier series*, Applied and Computational Harmonic Analysis, 47 (2019), pp. 585–606, https://doi.org/10.1016/j.acha.2017.11.001.
[18] R. M. Slevinsky and S. Olver, *Fasttransforms.jl*, https://github.com/JuliaApproximation/FastTransforms.jl.
[19] T.-J. Stieltjes, *Sur les polynômes de Legendre*, Annales de la Faculté des sciences de Toulouse : Mathématiques, 1e série, 4 (1890), pp. G1–G17.
[20] A. Townsend, M. Webb, and S. Olver, *Fast polynomial transforms based on Toeplitz and Hankel matrices*, Mathematics of Computations, 87 (2018), pp. 1913–1934.
[21] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, *Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, New York, NY, USA, 2013, Association for Computing Machinery, https://doi.org/10.1145/2503210.2503219.

Appendix.

A.1. Derivation of Eq. (4.3). In order to derive Eq. (4.3) we will be using the same variables and parameters as defined in Sec. 4. We start by assuming that the work array $\boldsymbol{w}(\gamma+1) \in \mathbb{R}^{b_{\gamma+1}\times 2\times M}$, defined prior to Eq. (4.1), is known, and the objective is to efficiently compute $\boldsymbol{w}(\gamma)$. To this end we first write Eq. (4.1) in index form for each parity as

$$w_n(\gamma,b,q) = \sum_{m=0}^{h_\gamma-1} T_n(Y_m^{\sigma,\gamma}) f_m^\sigma(\gamma,b,q), \quad \forall\, n \in \mathcal{I}_M, \tag{A.1}$$

with $Y_m^{\sigma,\gamma}$ defined in Eq. (3.25). Direct summation of (A.1) costs $4Mh_\gamma$ flops, which we intend to improve. We now split the reference interval $I=[-1,1]$ for a submatrix on level γ into 2 smaller subintervals

$$I(q) = [q-1, q], \quad q \in \{0,1\}, \tag{A.2}$$

such that $I = I(0) \cup I(1)$. The two subintervals are shown in Fig. 2. Next, we split the sum in Eq. (A.1) into 2 parts corresponding to the 2 new subintervals

$$w_n(\gamma,b,q) = \sum_{l\in\{0,1\}} \sum_{m=lh_{\gamma+1}}^{(l+1)h_{\gamma+1}-1} T_n(Y_m^{\sigma,\gamma}) f_m^\sigma(\gamma,b,q), \tag{A.3}$$

where $Y_m^{\sigma,\gamma} \in I(l)$, for all $lh_{\gamma+1} \le m < (l+1)h_{\gamma+1}$. The objective now is to replace terms on the right hand side of Eq. (A.3) with terms already computed on level $\gamma+1$. To this end we first recognise from Fig. 2 (or Eq. (3.14)) that

$$\{f_m^\sigma(\gamma+1, 2b+q+1, l)\}_{m=0}^{h_{\gamma+1}-1} = \{f_m^\sigma(\gamma,b,q)\}_{m=lh_{\gamma+1}}^{(l+1)h_{\gamma+1}-1}, \tag{A.4}$$

which can be used directly in Eq. (A.3). We get

$$w_n(\gamma,b,q)=\sum_{l\in\{0,1\}}\sum_{m=0}^{h_{\gamma+1}-1}T_n(Y^{\sigma,\gamma}_{m+lh_{\gamma+1}})f^{\sigma}_m(\gamma+1,2b+q+1,l), \tag{A.5}$$

where we see that for given l the Chebyshev basis functions are using coordinates only for half the regular interval since $Y^{\sigma,\gamma}_{m+lh_{\gamma+1}}\in I(l)$ for $l\in\{0,1\}$ and $m\in\mathcal{I}_{h_{\gamma+1}}$.

In order to map the half-interval Chebyshev basis functions in (A.5) to the full-interval basis functions used on level $\gamma+1$ we introduce

$$T_n((Y+2q-1)/2)=\sum_{k=0}^{n}b^{(q)}_{nk}T_k(Y),\quad Y\in I,\quad q\in\{0,1\}, \tag{A.6}$$

where $(Y+2q-1)/2\in I(q)$ for $q\in\{0,1\}$. We can compute the matrices $\boldsymbol{B}(q)=(b^q_{nk})_{n,k=0}^{M-1}$ using the orthogonality of Chebyshev polynomials or, alternatively, from the binomial theorem and basis changes, as shown in Appendix A.3. The matrices are lower triangular since the half-interval Chebyshev polynomials $T_n((Y+2q-1)/2)$ are still polynomials in $\mathbb{P}_n(I(q))$ and for the first 4 rows we get

$$\boldsymbol{B}(0)=\begin{bmatrix}1&0&0&0&0&\cdots\\ -\frac{1}{2}&\frac{1}{2}&0&0&0&\cdots\\ -\frac{1}{4}&-1&\frac{1}{4}&0&0&\cdots\\ \frac{1}{4}&\frac{3}{8}&-\frac{3}{4}&\frac{1}{8}&0&\cdots\\ \vdots&\vdots&\vdots&\vdots&\vdots&\ddots\end{bmatrix},\quad \boldsymbol{B}(1)=\begin{bmatrix}1&0&0&0&0&\cdots\\ \frac{1}{2}&\frac{1}{2}&0&0&0&\cdots\\ -\frac{1}{4}&1&\frac{1}{4}&0&0&\cdots\\ -\frac{1}{4}&\frac{3}{8}&\frac{3}{4}&\frac{1}{8}&0&\cdots\\ \vdots&\vdots&\vdots&\vdots&\vdots&\ddots\end{bmatrix}. \tag{A.7}$$

Going back to Eq. (A.5) we can now use Eq. (A.6) on the right hand side, and since $Y^{\sigma,\gamma}_{m+lh_{\gamma+1}}=(Y^{\sigma,\gamma+1}_m+2l-1)/2$ for $m\in\mathcal{I}_{h_{\gamma+1}}$ and $l\in\{0,1\}$ we have

$$T_n(Y^{\sigma,\gamma}_{m+lh_{\gamma+1}})=\sum_{k=0}^{n}b^{(l)}_{nk}T_k(Y^{\sigma,\gamma+1}_m),\quad \text{for } l\in\{0,1\},\, m\in\mathcal{I}_{h_{\gamma+1}}. \tag{A.8}$$

We obtain

$$w_n(\gamma,b,q)=\sum_{l\in(0,1)}\sum_{k=0}^{n}b^{(l)}_{nk}\sum_{m=0}^{h_{\gamma+1}-1}T_k(Y^{\sigma,\gamma+1}_m)f^{\sigma}_m(\gamma+1,2b+q+1,l), \tag{A.9}$$

and realise that the last sum in Eq. (A.9) has already been computed on level $\gamma+1$ with the result stored in $\boldsymbol{w}(\gamma+1,2b+q+1,l)$. As such we can rearrange and simplify into

$$w_n(\gamma,b,q)=\sum_{l\in\{0,1\}}\sum_{k=0}^{n}b^{(l)}_{nk}w_k(\gamma+1,2b+q+1,l), \tag{A.10}$$

which in matrix form becomes Eq. (4.3).

A.2. Computing Eq. (4.3) efficiently. Algorithm A.1 computes Eq. (4.3) in M^2+2M operations using the fact that $\boldsymbol{B}(0)$ and $\boldsymbol{B}(1)$ differ only in the sign of the odd diagonals, see App. A.3.

Algorithm A.1 A fast computation of Eq. (4.3) costing $M^2 + 2M$ flops, attributed to one lower triangular matrix-vector product plus setting up the work array $\boldsymbol{z}$.

function EQ43($\boldsymbol{w}^0, \boldsymbol{w}^1, \boldsymbol{B}$)
input $\boldsymbol{w}^0$: array $\in \mathbb{R}^M$ ▷ $\boldsymbol{w}(\gamma, b, q)$ in Eq. (4.3)
input $\boldsymbol{w}^1$: array $\in \mathbb{R}^{2\times M}$ ▷ $\boldsymbol{w}(\gamma+1, 2b+q+1)$ in Eq. (4.3)
input $\boldsymbol{B}$: $\mathbb{R}^{M\times M}$ ▷ $\boldsymbol{B}(0)$ in Eq. (4.3)
work array $\boldsymbol{z}$: $\mathbb{R}^{2\times M}$
    $M \leftarrow$ **len** $\boldsymbol{w}^0$
    for $i \leftarrow 0, M-1$ **do**
        $z_{0,i} \leftarrow w^1_{0,i} + w^1_{1,i}$
        $z_{1,i} \leftarrow w^1_{0,i} - w^1_{1,i}$
    end for
    $n \leftarrow 0$
    while $n < M$ **do**
        $k \leftarrow n$ **mod** 2
        for $i \leftarrow 0, M-n-1$ **do**
            $w^0_{i+n} \leftarrow w^0_{i+n} + b_{i+n,i} z_{k,i}$
        end for
        $n \leftarrow n+1$
    end while
    return $\boldsymbol{w}^0$
end function

A.3. Computing the matrices $\boldsymbol{B}(l)$ in Eq. (4.3) exactly. We compute the matrices using a simple basis change and the binomial theorem. We have that

$$x^n = \sum_{j=0}^{n} w_{nj} T_j(x), \quad n \in \mathcal{I}_M, \tag{A.11}$$

or with $\underline{\boldsymbol{x}} = (x^j)_{j=0}^{M-1}$, $\boldsymbol{W} = (w_{nj})_{n,j=0}^{M-1}$ and $\underline{\boldsymbol{T}} = (T_j(x))_{j=0}^{M-1}$

$$\underline{\boldsymbol{x}} = \boldsymbol{W}\underline{\boldsymbol{T}}. \tag{A.12}$$

The lower triangular coefficient matrix $\boldsymbol{W}$, where only the even diagonals are nonzero, is given as [10]

$$w_{nj} = \begin{cases} \frac{2}{c_j 2^n}\binom{n}{\frac{n-j}{2}}, & n-j \text{ even and } j \le n, \\ 0, & \text{otherwise.} \end{cases} \tag{A.13}$$

The inverse matrix $\boldsymbol{W}^{-1}$ is also lower triangular and given as

$$w^{-1}_{nj} = \begin{cases} 1, & n = j = 0, \\ \frac{n(-1)^{\frac{n-j}{2}} 2^j}{n+j}\binom{\frac{n+j}{2}}{\frac{n-j}{2}}, & n-j \text{ even}, n > 0 \text{ and } j \le n, \\ 0, & \text{otherwise,} \end{cases} \tag{A.14}$$

rewritten here from [10] such that $\underline{\boldsymbol{T}} = \boldsymbol{W}^{-1}\underline{\boldsymbol{x}}$.

The matrices $\boldsymbol{B}(l)$ are used to express the half-interval Chebyshev polynomials $T_k((x+2l-1)/2)$ in terms of $\{T_n(x)\}_{n=0}^k$, where $x \in [-1,1]$ and $(x+2l-1)/2 \in [l-1,l]$ for $l \in \{0,1\}$. The binomial theorem tells us that

$$\left(\frac{x+2l-1}{2}\right)^n = \sum_{j=0}^{n} v_{nj}^{(l)} x^j, \tag{A.15}$$

where

$$v_{nj}^{(l)} = \begin{cases} \binom{n}{n-j} \frac{(2l-1)^{n-j}}{2^n}, & j \leq n, \\ 0, & \text{otherwise.} \end{cases} \tag{A.16}$$

In matrix form, with $\overline{\boldsymbol{x}}(l) = \{((x+2l-1)/2)^j\}_{j=0}^{M-1}$ and $\boldsymbol{V}(l) = (v_{nj}^{(l)})_{n,j=0}^{M-1}$, we get

$$\overline{\boldsymbol{x}} = \boldsymbol{V}\underline{\boldsymbol{x}}. \tag{A.17}$$

From Eq. (A.11) we have

$$\overline{\boldsymbol{x}} = \boldsymbol{W}\overline{\boldsymbol{T}}, \tag{A.18}$$

where $\overline{\boldsymbol{T}}(l) = \{T_j((x+2l-1)/2)\}_{j=0}^{M-1}$. Setting now Eq. (A.17) equal to (A.18) and using Eq. (A.12) we get

$$\boldsymbol{V}\boldsymbol{W}\underline{\boldsymbol{T}} = \boldsymbol{W}\overline{\boldsymbol{T}}, \tag{A.19}$$

and thus

$$\overline{\boldsymbol{T}} = \boldsymbol{W}^{-1}\boldsymbol{V}\boldsymbol{W}\underline{\boldsymbol{T}}. \tag{A.20}$$

Comparing Eq. (A.20) to Eq. (A.6) we get that $\boldsymbol{B}(l) = \boldsymbol{W}^{-1}\boldsymbol{V}(l)\boldsymbol{W}$. From Eq. (A.16) it is obvious that $\boldsymbol{V}(0)$ and $\boldsymbol{V}(1)$ differ only in the sign of the odd diagonals. Since $\boldsymbol{W}$ and $\boldsymbol{W}^{-1}$ decouple the odd and even components (every odd diagonal is zero) this means that also $\boldsymbol{B}(0)$ and $\boldsymbol{B}(1)$ will differ only in the sign of the odd diagonals. Yet another interesting computational feature of the $\boldsymbol{B}(l)$ matrices is that for $M = 18$ they can be represented exactly in double precision since all items are rational numbers with the denominator being a small power of 2.