Fast Multipole: It’s all about Adding Functions in Finite Precision

Alan Edelman and Per-Olof Persson

April 26, 2004

DRAFT - This is Work in Progress

1 Introduction

This paper contains a new algorithmic idea, a number of novel approaches, and also the distillation of eleven years of teaching graduate students at MIT about the Fast Multipole Method. Regarding the latter, there are many excellent introductions to the Fast Multipole Method [2], [7], [3], including the original thesis of Greengard [6].

Our title indicates an approach to Fast Multipole that exposes and explores the abstractions underlying the algorithm. In order to remove the extraneous layers so as to reach the simple core we hope that our title summarizes the two critical levels:

1. The high level (HOW we operate): Traditional FMM adds functions through the use of a parallel prefix style tree-based algorithm. This algorithm organizes the order and the data-structures of the adding, but does not provide any specification of the addition itself.
2. The low level (WHAT we operate on): On a computer we add a finite precision representation of functions. How we accomplish this representation and the corresponding addition is discussed at this level.

To some extent we can decouple the two levels, but not completely as choices made at the high level lead to constraints at the low level.

We believe that the basic ideas are best understood by studying the 1-D problem, and we focus on the simple potential function $1/x$ instead of the more common alternatives $\log|x|$ and $1/|x|$. For details on higher dimensions and other potentials we refer to other work.

1.1 The high level: A five column combination of “building blocks”

By identifying key pieces of the puzzle, we can imagine combinations that place the high level part of FMM into a bigger context. We illustrate in Table 1 the five key ingredients that spotlight

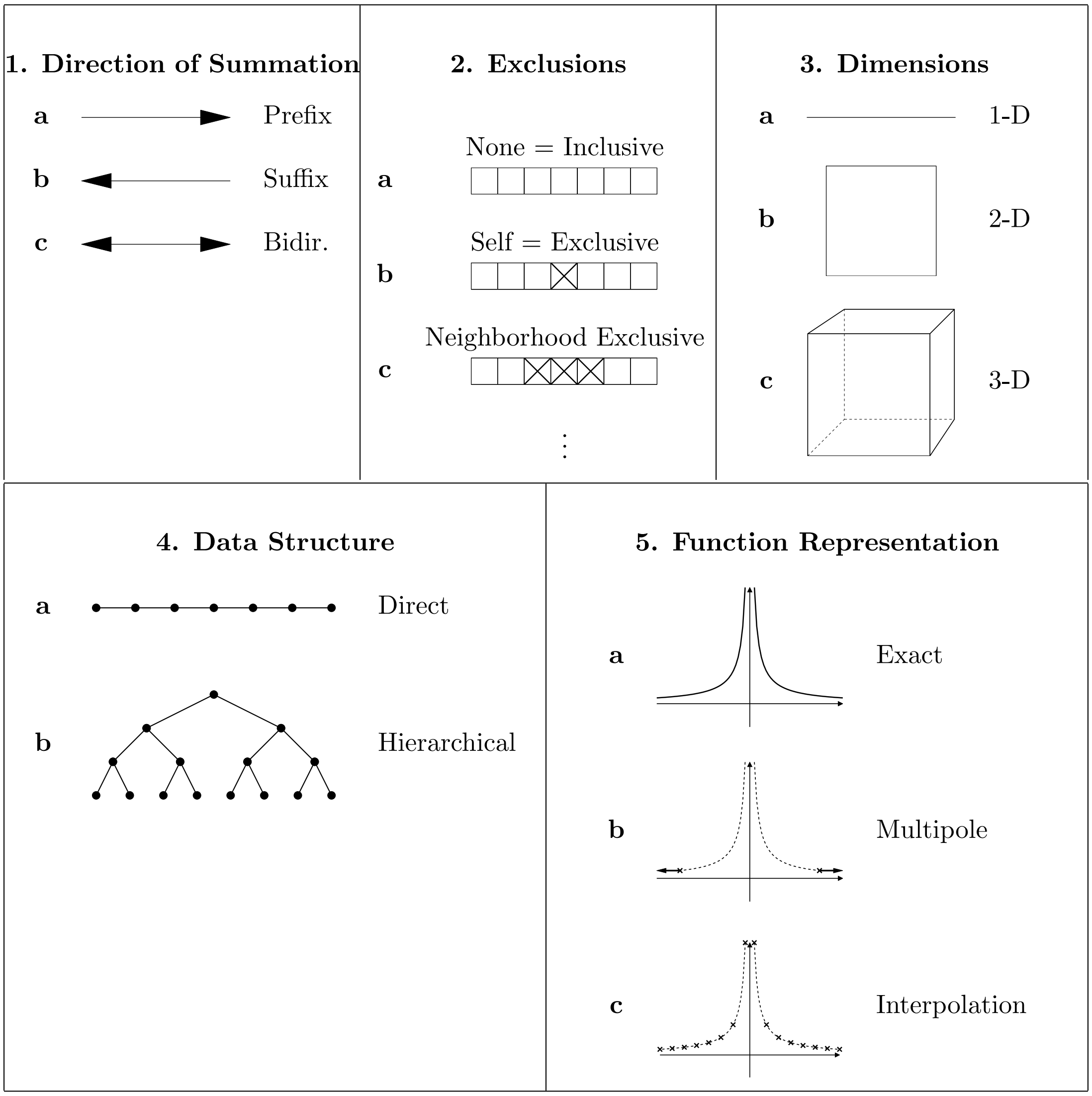


Table 1: The structure of the family of algorithms we have identified. The traditional FMM corresponds to the choices 1c+2c+3(any)+4b+5b.

(somewhat arbitrarily) our abstraction. You are meant to pick one item from each column to build an algorithm. The first four columns are discrete, the fifth is continuous. Traditional 2-D FMM might be characterized as

$$\textbf{2-D FMM:} \qquad \mathbf{1c + 2c + 3b + 4b + 5b}$$

We think of the “a” choices as being perhaps the simplest in each column. An algorithm familiar to the parallel algorithm community known as “parallel prefix” might be characterized as

$$\textbf{Parallel Prefix:} \qquad \mathbf{1a + 2a + 3a + 4b + 5a}$$

Already an interesting paradox is exposed. FMM is a powerful algorithm on a non-parallel computer. A serial prefix would use 4a and not 4b. Can there be an FMM using 4a and not 4b? This paper indicates how to implement such an algorithm, and with appropriate function representations this could lead to a new direct version of the FMM.

1.2 The low level: Column 5 and the analogy with the floating point representation of numbers and the IEEE standard for arithmetic

May we begin with floating point numbers? We wish to gently guide the reader by analogy one rung up the abstraction ladder towards the finite discretization of functions.

Somewhere close to the hardware level, somebody worries about the representation and addition of real numbers. Design choices include how many digits or bits, and even whether to use binary, decimal, or hexadecimal. In floating point, we decouple the exponent from the mantissa. Interval arithmetic opens up a whole new chapter.

At this low level, we must worry about operations on these representations. As children we learn what it means to add, yet hardware designers must carefully consider implementation details of adding in floating point and, say, correctly rounding the result. The loss is known as rounding error. At this level, addition is no longer the trivial operation we remember from childhood but rather the specialty of hardware designers.

We are ready for our analogy. By high school, we learn what it means to add functions. We learn that $\sin^2 x + \cos^2 x = 1$ and $\log x + \log x = \log x^2$ yet FMM algorithm designers must carefully consider implementation details of adding functions with a finite representation and correctly representing the result. The loss is known as “approximation error”. At this level, addition is no longer the familiar territory we remember from high school but rather the specialty of FMM experts.

Even those familiar with FMM may find the analogy compelling. The functions that we add in FMM might perhaps look something like $f(x) = q/(x-c)$ or $f(x) = q\log(x-c)$ or even $f(x) = q\cot(x-c)$, but probably not $f(x) = \exp(-(x-c)^2)$. We tend to specialize on functions with a “slow decay” away from a singularity; the “exp” example has a very fast decay. However, $\exp(-(x-c)^2/h)$ can be a good candidate for FMM if h is small enough – these are the fast Gauss transforms and can work wonders in high dimensions.

We invite readers to continue to Section 3 for an in-depth study of some choices available to an FMM designer. Meanwhile for reference we have a table to help the reader with the analogy between floating point numbers and discretized functions:

	Floating Point Numbers	**Discretized Functions**
Theoretical Objects under consideration	Real Numbers	Smooth Functions with preordained properties
Some Rep Choices	Binary, Decimal, Hex	Multipole, Polynomial, Virtual Charges
Some sizes	Single Prec = 23 mantissa bits Double Prec = 52 mantissa bits Any number of bits possible	We can take "p" coefficients
Meaning of Rounding	Some number of bits are accurate or reasonably close	The function is accurate at some inputs or reasonably close
Rounding choices	Round to nearest, deal with ties sensibly	Interpolate, approximate in some sensible norm
Adding	Intricate algorithm to give an answer within a certain tolerance or as in IEEE the correctly rounded answer assuming the inputs are accurate	Intricate algorithm to give a representation of a function that is within a certain tolerance of the true sum

The analogy is perfect. Let us push the analogy if we might, with numbers. In mathematics we have exact numbers such as $1/2, 1/3, 1/10, \sqrt{2}$ and π. Only the first of these numbers is represented exactly on most modern computers. In IEEE double precision, numbers are represented as 64 bit strings, with 52 bits reserved for the mantissa. Even 1/10 has an infinite binary expansion and can not be represented exactly in binary with finitely many bits. Anticipating functions, we describe the familiar representation of numbers in a suggestive manner. For convenience, we normalize, a number f by assuming f is a real number in $[1, 2)$. Any such number may be written in binary as $1.b_1b_2b_3, \ldots$, i.e., we have a "function" from the integers $i = 1, 2, 3, \ldots$ to the binary numbers 0 and 1. When rounding to double precision this function is exact only on the interval $i = [1, 52]$. The reader should read the last sentence again. Notice how we pointed out that the simple idea of rounding is being viewed (perhaps for pedagogical purposes only) as a function that is accurate on 52 bits.

As another little tidbit to entice readers to move onto Section 3, we will point out a novel application of the Lanczos algorithm to create virtual charges that can give an interesting alternative to the multipole series. Most multipole approaches to approximation are linear. Almost as good as linear, by our viewpoint, is an eigenvalue algorithm. Please turn to Section 3.

2 Parallel and Serial Addition

This section is mostly about recursive operators known to computer scientists as the parallel prefix operator or the "scan" operator. The prefix sum operator $y_k = \sum_{i=1}^{k} x_i$ might be recognized in the forms shown in Table 2.

As the name suggests, this operator has found many uses in parallel computing from the early application of parallel carry look-ahead addition in the 1800's to implementations on modern supercomputers. Our setting is not necessarily parallel computing but the underlying tree implementation is important for our case of exclusive add.

The prefix sum operator $y_k = \sum_{i=1}^{k} x_i$

Language	**Notation**
Linear Algebra	$y = \begin{pmatrix} 1 & & 0 \\ \vdots & \ddots & \\ 1 & \dots & 1 \end{pmatrix} x$
MPI	`MPI_SCAN(x,y,length,datatype,MPI_SUM,...)`
MATLAB	`y=cumsum(x)`
APL	`y← +\x`
High Performance Fortran	`Y=SUM_PREFIX(X,...)`

Table 2: Notation for the prefix sum operator in different languages.

This section of the paper contains most of the algorithmic underpinnings of the FMM, without any distractions related to function approximation. By decoupling the addition (which is far more general) from the functions, the reader can understand the algorithm unencumbered by the baggage of function representations.

2.1 Columns 1 and 2: What we are adding

Columns 1 and 2 of Table 1 are easy enough to understand. They explain at the vector level what we wish to add. Technically we are given a vector $x = (x_1, \ldots, x_n)$ and an associative operator "+". The operations are $y_k = \sum_{i \in I} x_i$ where the intervals are:

	2a: Inclusive	**2b**: Exclusive	**2c**: Neighborhood-exclusive
1a: Prefix	$1 : k$	$1 : (k-1)$	$1 : (k-2)$
1b: Suffix	$k : n$	$(k+1) : n$	$(k+2) : n$
1c: Bidirectional	$1 : n$	$1 : (k-1)$ & $(k+1) : n$	$1 : (k-2)$ & $(k+2) : n$

Thus inclusive prefix sum is $y_k = \sum_{i \le k} x_i$ and exclusive bidirectional sum is $y_k = \sum_{i \ne k} x_i$.

The nine operations in linear algebra notation can all be written as $y = Mx$ where $M =$

	2a: Inclusive	**2b**: Exclusive	**2c**: Neighborhood-exclusive
1a: Prefix	$\begin{pmatrix} 1 & . & . & . \\ 1 & 1 & . & . \\ 1 & 1 & 1 & . \\ 1 & 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} . & . & . & . \\ 1 & . & . & . \\ 1 & 1 & . & . \\ 1 & 1 & 1 & . \end{pmatrix}$	$\begin{pmatrix} . & . & . & . \\ . & . & . & . \\ 1 & . & . & . \\ 1 & 1 & . & . \end{pmatrix}$
1b: Suffix	$\begin{pmatrix} 1 & 1 & 1 & 1 \\ . & 1 & 1 & 1 \\ . & . & 1 & 1 \\ . & . & . & 1 \end{pmatrix}$	$\begin{pmatrix} . & 1 & 1 & 1 \\ . & . & 1 & 1 \\ . & . & . & 1 \\ . & . & . & . \end{pmatrix}$	$\begin{pmatrix} . & . & 1 & 1 \\ . & . & . & 1 \\ . & . & . & . \\ . & . & . & . \end{pmatrix}$
1c: Bidirectional	$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} . & 1 & 1 & 1 \\ 1 & . & 1 & 1 \\ 1 & 1 & . & 1 \\ 1 & 1 & 1 & . \end{pmatrix}$	$\begin{pmatrix} . & . & 1 & 1 \\ . & . & . & 1 \\ 1 & . & . & . \\ 1 & 1 & . & . \end{pmatrix}$

2.2 Hierarchical algorithms (4b): Exclusive Add

2.2.1 Six Notations, One Algorithm

Starting with a vector x of length $n = 2^d$ (for simplicity only), consider the operation of forming $y_i = \sum_{j \le i} x_j$. This is a prefix add. For example, when $n = 8$ in matrix notation, we have that

$$y = \begin{pmatrix} 1 & . & . & . & . & . & . & . \\ 1 & 1 & . & . & . & . & . & . \\ 1 & 1 & 1 & . & . & . & . & . \\ 1 & 1 & 1 & 1 & . & . & . & . \\ 1 & 1 & 1 & 1 & 1 & . & . & . \\ 1 & 1 & 1 & 1 & 1 & 1 & . & . \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & . \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} x, \tag{1}$$

(we use dots to represent 0's). FMM focuses on minor variations, such as the exclusive add $y_i = \sum_{j \ne i} x_j$ and even more importantly the neighborhood exclusive add $y_i = \sum_{|j-i|>1} x_j$:

$$y = \begin{pmatrix} . & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & . & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & . & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & . & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & . & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & . & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & . & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & . \end{pmatrix} x \quad \text{and} \quad y = \begin{pmatrix} . & . & 1 & 1 & 1 & 1 & 1 & 1 \\ . & . & . & 1 & 1 & 1 & 1 & 1 \\ 1 & . & . & . & 1 & 1 & 1 & 1 \\ 1 & 1 & . & . & . & 1 & 1 & 1 \\ 1 & 1 & 1 & . & . & . & 1 & 1 \\ 1 & 1 & 1 & 1 & . & . & . & 1 \\ 1 & 1 & 1 & 1 & 1 & . & . & . \\ 1 & 1 & 1 & 1 & 1 & 1 & . & . \end{pmatrix} x. \tag{2}$$

The exclusive add is the essential operation of FMM, though FMM usually implements the "neighborhood exclusive" add which excludes each item and its nearest neighbors. We will say

		2a: Inclusive	**2b**: Exclusive	**2c**: Neighborhood-exclusive
1a	i even	$y_i = w_{i/2}$	$y_i = w_{i/2} + x_{i-1}$	$y_i = w_{i/2} + x_{i-2}$
Prefix	i odd	$y_i = w_{(i+1)/2} + x_i$	$y_i = w_{(i+1)/2} + x_i$	$y_i = w_{(i+1)/2} + x_{i-3} + x_{i-2}$
1b	i even	$y_i = w_{i/2} + x_i$	$y_i = w_{i/2} + x_i$	$y_i = w_{i/2} + x_{i+2} + x_{i+3}$
Suffix	i odd	$y_i = w_{(i+1)/2}$	$y_i = w_{(i+1)/2} + x_{i-1}$	$y_i = w_{(i+1)/2} + x_{i+2}$
1c	i even	$y_i = w_{i/2}$	$y_i = w_{(i+1)/2}$	$y_i = w_{i/2} + x_{i+2} + x_{i+3} + x_{i-2}$
Bidirectional	i odd	$y_i = w_{(i+1)/2} + x_{i-1}$	$y_i = w_{(i+1)/2} + x_{i+1}$	$y_i = w_{(i+1)/2} + x_{i+2} + x_{i-3} + x_{i-2}$

Table 3: Variations on the prefix sum update formula for different directions and different treatment of the "self-term" and its neighbors.

more about this in Section 2.3. We begin with five notations for the same algorithm. Readers will see that this is not overkill.

How should we implement an exclusive add? Certainly not with $O(n^2)$ operations. We might consider forming `sum(x)-x`. At least this is $O(n)$ operations, but in floating point, if one element of x is very large, the result would be inaccurate. Here is an example using the exclude function that follows in Notation II. If the inputs are scaled very differently, this algorithm is more accurate than the naive choice. This is not an idol curiosity, this is intrinsic to why FMM must be organized the way it is. We do not wish to add in and later subtract out singularities or large numbers. The MATLAB example below shows that this can give problems even with ordinary floating point numbers:

```
>> x=[1e100 1 1 1 1 1 1 1]; y=exclude(x);y(1)
ans =
     7 % Correct Answer
>> sum(x)-x(1)
ans =
     0 % Wrong Answer
```

Algorithm Notation I: Pseudocode

A clever approach uses a recursive algorithm built from three steps:

1. Pairwise Adds ($w_i = x_{2i} + x_{2i-1}$)

2. Recursive Exclusive Add on w

3. Update Adds $y_i = w_{i/2} + x_{i-1}$ (even i) or $y_i = w_{(i+1)/2} + x_{i+1}$ (odd i)

We mentioned that prefix is a minor variation. Merely change line 3 to $y_i = w_{i/2}$ (even i) or $y_i = w_{(i+1)/2} + x_i$ (odd i). Indeed many variations are possible by tinkering with line 3, see Table 3.

Algorithm Notation II: MATLAB

The following simple MATLAB function implements the exclude algorithm 1c+2b (on scalars or even symbolic functions):

```
function y=exclude(x)
n=length(v);
if n==1, y=0; else
  w=x(1:2:n)+x(2:2:n);                      % Pairwise adds
  w=exclude(w);                             % Recur
  y(1:2:n)=w+x(2:2:n); y(2:2:n)=w+x(1:2:n);  % Update Adds
end
```

For example, if the inputs are integers:

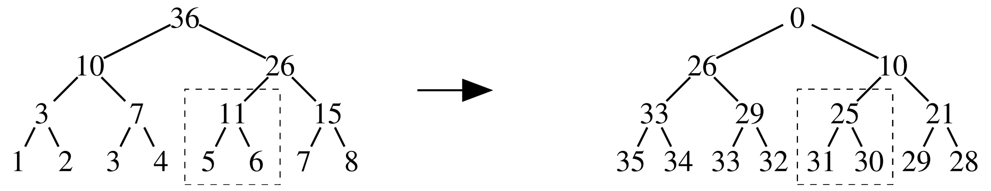
```
>> exclude(1:8)
ans =
    35    34    33    32    31    30    29    28
```

The symbolic example (requires the MATLAB Symbolic Toolbox) also works on functions without any code change. Here our input vector is

$$\left(\frac{1}{t-1}, \frac{1}{t-2}, \frac{1}{t-3}, \frac{1}{t-4}\right)$$

```
>> syms t
>> exclude(1./(t-(1:4)))
ans =
[ 1/(t-2)+1/(t-3)+1/(t-4), 1/(t-1)+1/(t-3)+1/(t-4),
  1/(t-4)+1/(t-1)+1/(t-2), 1/(t-3)+1/(t-1)+1/(t-2)]
```

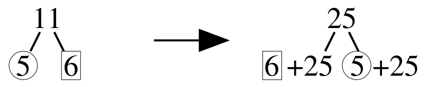
Algorithm Notation III: Trees



Up the tree (Pairwise adds)

Down the tree (Update adds with sibling swap)

Readers will have no trouble understanding the “Up the tree” diagram on the left. Going down the tree, the original values of two siblings are swapped and added to the new value of their parent, e.g.



Update adds with sibling swap

Algorithm Notation IV: Matrices

In Matrix notation, the recursive algorithm is based on a decomposition that represents a rank n matrix as the sum of a rank $n/2$ matrix and a block diagonal matrix.

$$\begin{pmatrix} . & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & . & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & . & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & . & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & . & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & . & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & . & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & . \end{pmatrix} = \begin{pmatrix} 1 & . & . & . \\ 1 & . & . & . \\ . & 1 & . & . \\ . & 1 & . & . \\ . & . & 1 & . \\ . & . & 1 & . \\ . & . & . & 1 \\ . & . & . & 1 \end{pmatrix} \underbrace{\begin{pmatrix} . & 1 & 1 & 1 \\ 1 & . & 1 & 1 \\ 1 & 1 & . & 1 \\ 1 & 1 & 1 & . \end{pmatrix}}_{\text{Recursion}} \underbrace{\begin{pmatrix} 1 & . & . & . \\ 1 & . & . & . \\ . & 1 & . & . \\ . & 1 & . & . \\ . & . & 1 & . \\ . & . & 1 & . \\ . & . & . & 1 \\ . & . & . & 1 \end{pmatrix}^T}_{\text{Pairwise Adds}} + \underbrace{\begin{pmatrix} . & 1 & . & . & . & . & . & . \\ 1 & . & . & . & . & . & . & . \\ . & . & . & 1 & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & . & 1 \\ . & . & . & . & . & . & 1 & . \end{pmatrix}}_{\text{Update adds}} . \tag{3}$$

Algorithm Notation V: Kronecker Products

Of course, one can apply the decomposition recursively to the 4×4 matrix above, and so on. In Kronecker (or tensor product) notation, we have that

$$A_8 = \left(I_4 \otimes \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right) A_4 \left(I_4 \otimes \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right)^T + I_4 \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} . \tag{4}$$

If we wanted to, we could construct whole families of recursive matrix factorizations. All that is needed is to change the "update add" step. (Perhaps you can imagine changing the pairwise add too.)

Algorithm Notation VI: Data Arrays

It is of interest to graphically view the "update adds" of notation I (or equivalently the down the tree part of notation III, etc) as we do in Figure 1. The figure may be easier to figure out yourself, than for us to articulate what it is saying in words. One can see that upon-completion the "fill-in" in each matrix matches that of Equation (2).

The figure at each time step indicates which entries are folded in as we move down the tree. In the top row we see the sibling swap. A contiguous black row, say black elements in row i with contiguous column entries $j_1 : j_2$, means that the answer in index i has now folded in the sum from j_1 to j_2 by adding in one number. The gray elements are numbers that have been folded in from previous steps.

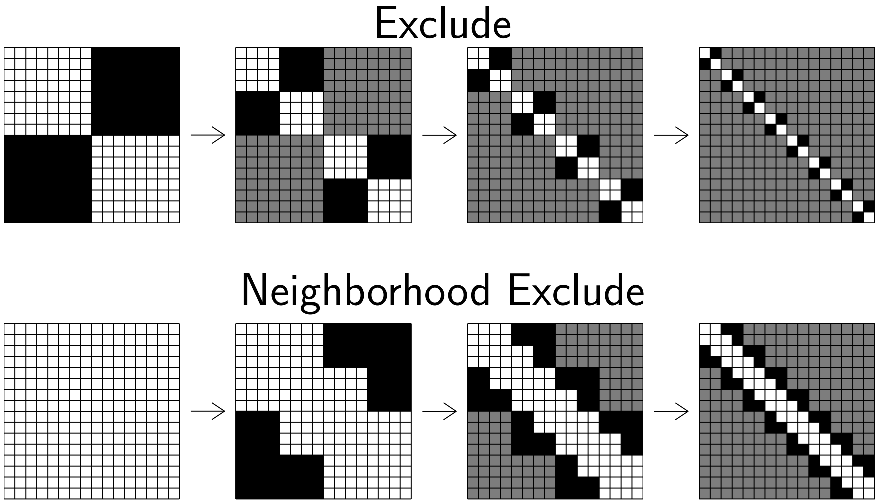


Figure 1: Illustration of the exclude and the neighborhood exclude functions. The white squares correspond to data that have not been added yet, the gray squares to data that has been added, and the black ones to data that is being added in the current step. Note that the "large" black squares indicates the result of one single sum that is being used multiple times.

2.3 Neighborhood Exclusive Add (2c)

In the previous section we devoted a great deal of attention to the exclusive add (2b). The true FMM is a neighborhood exclude add as in $y = M_8 x$ using the matrix below.

$$M_8 = \begin{pmatrix} . & . & 1 & 1 & 1 & 1 & 1 & 1 \\ . & . & . & 1 & 1 & 1 & 1 & 1 \\ 1 & . & . & . & 1 & 1 & 1 & 1 \\ 1 & 1 & . & . & . & 1 & 1 & 1 \\ 1 & 1 & 1 & . & . & . & 1 & 1 \\ 1 & 1 & 1 & 1 & . & . & . & 1 \\ 1 & 1 & 1 & 1 & 1 & . & . & . \\ 1 & 1 & 1 & 1 & 1 & 1 & . & . \end{pmatrix}. \tag{5}$$

We believe the reader can fill in the remaining notations, we feel satisfied to include only the MATLAB code for neighborhood exclude add.

Algorithm Notation II: MATLAB

```
function y=exclude_neighborhood(x)

n=length(x);
if n==2
  y=[0*x(1) 0*x(2)];
```

```
else
  w=exclude_neighborhood(x(1:2:n)+x(2:2:n));
  if n>4,
    y(3:2:n-3)=x(5:2:n-1)+w(2:n/2-1)+x(6:2:n)+x(1:2:n-5);
    y(4:2:n-2)=x(2:2:n-4)+w(2:n/2-1)+x(1:2:n-5)+x(6:2:n);
  end
  y(1)=x(3)+w(1)+x(4);  y(2)=w(1)+x(4);
  y(n-1)=w(n/2)+x(n-3);  y(n)=x(n-2)+w(n/2)+x(n-3);
end
```

2.4 A direct algorithm (4a)

It hardly is worth spelling out how to perform any of these algorithms in serial. Inclusive Prefix is simply a cumulative addition from the left. Exclusive bidirectional add suggests forming the exclusive prefix and exclusive suffix and then adding. (We do not wish to allow subtraction or an easier algorithm "`sum(x)-x`" comes to mind.) The contrast between the simplicity of the direct algorithm as compared to the hierarchical algorithm is evident. For the 1-D problem, we can express the neighborhood exclusive sum as:

$$y_i^{\mathrm{ne}} = \sum_{|j-i|>1} x_j = y_{i-2}^{\mathrm{p}} + y_{i+2}^{\mathrm{s}}, \tag{6}$$

where the prefix sum $y_i^{\mathrm{p}} = \sum_{k=1}^{i} x_i$ and the suffix sum $y_i^{\mathrm{s}} = \sum_{k=i}^{n} x_i$ are computed using cumulative sums with a total of $2n$ operations (we set $y_i^{\mathrm{p}} = 0$ for i outside the domain, to avoid special cases for the boundaries). This algorithm has several advantages over the tree-based algorithm: Easier to implement and to understand, fewer total operations, and no restrictions on the number of elements n (such as being a power of 2). The algorithm can be expressed in MATLAB notation as:

```
wpre(1)=0; for i=3:n,  wpre(i)=x(i-2)+wpre(i-2);         end
wsuf(n)=0; for i=3:n,  wsuf(n-i+1)=x(n-i+3)+wsuf(n-i+3); end
y=wpre+wsuf;
```

We can extend our direct serial algorithm (no trees!) to higher dimensions. The 2-D algorithm is illustrated in Figure 2. We compute

$$y_{ij}^{\mathrm{ne}} = \sum_{\substack{|k-i|>1 \\ |\ell-j|>1}} x_{k\ell} = y_{i-2,j+1}^{\mathrm{pp}} + y_{i+1,j+2}^{\mathrm{ps}} + y_{i+2,j-1}^{\mathrm{ss}} + y_{i-1,j-2}^{\mathrm{sp}}. \tag{7}$$

Here, the superindices indicate prefix or suffix sums in each of the two dimensions, for example $y_{ij}^{\mathrm{pp}} = \sum_{k=1}^{i} \sum_{\ell=1}^{j} x_{k\ell}$, etc. These can again be constructed in linear time by successive cumulative sums. Note how the four terms "wrap around" the center cell i, j to include the whole region $|k-i| > 1, |\ell - j| > 1$.

The process generalizes to any dimension D, where 2^D prefix and suffix sums are formed and added in a similar way. For this to be possible, we need a construction that fills a block with 2^D smaller blocks, each aligned with one of the corners. One solution is shown in Figure 3

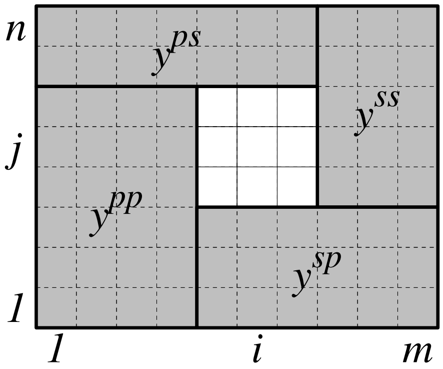


Figure 2: The non-hierarchical neighborhood exclusive add in 2-D. The region $|k-i| > 1, |\ell - j| > 1$ is built up by the prefix and suffix sums in (7), corresponding to the four blocks in the figure.

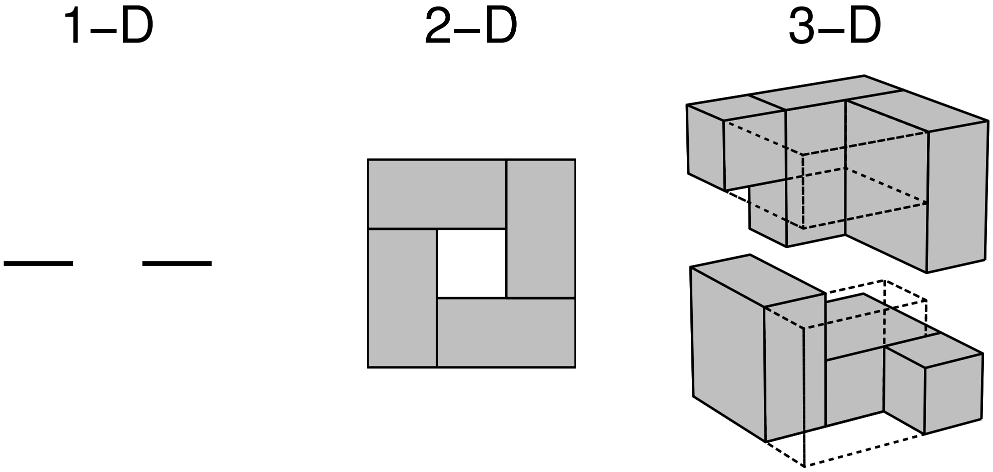


Figure 3: The construction of exclusive sums in $D = 1, 2, 3$ dimensions. The full region except for one block in the middle is covered by 2^D blocks – one for each corner.

for $D = 1, 2, 3$. We would like to thank Erik and Martin Demaine for showing us how to solve the problem in 4-D and 5-D.

While this algorithm is very appealing when adding numbers, it requires some extra thought to apply it to the full FMM problem (ordinary multipoles expansions do not seem to work because of the convergence radius, see next section). Nevertheless, we believe that there is potential in here for a new direct FMM, which might be simpler and more efficient. In Section 4 we describe some of our initial experiments in this direction.

2.5 The Cauchy Principal Value Integral as a Continuous Exclusive Add

Perhaps as a further justification of the "exclusive add" as a mathematical operator with a rich history we mention ever so briefly the Cauchy principal value integral.

Some readers may already be familiar with the Cauchy Principal Value integral

$$h(z) = PV \int_a^b \frac{f(x)}{x - z} dx, \ z \in (a, b).$$

It is very much a continuous exclusive add. The definition of the principal value recognizes the singularity when x is near z:

$$h(z) \equiv \lim_{\epsilon \to 0} \int_{[a,b]-(z-\epsilon, z+\epsilon)} \frac{f(x)}{x - z} dx.$$

In many ways, multipole is avoiding the local singularity through the exclusive add.

3 Finite Representations of Functions

3.1 Approximating Functions

Part of any function approximation scheme is a choice of representative functions. This section highlights polynomials, multipole series, and virtual charges as three choices among the infinitely many one can imagine.

One chooses the number p in advance, then we approximate functions as closely as possible with a vector of length p. Would it be too much of a stretch to say we are "rounding" functions to p coefficients?

Any p dimensional vector space of functions can be a good candidate for an approximation space. This means that we can add two functions or multiply by a scalar and stay within the space.

Given the choice of vector space, it is usual a computationally infeasible problem to find the best approximation, but often truncated series expansions or interpolation can work well. Some readers may know that most elementary functions (such as exp and sin) are not computed in practice by the Taylor series seen in calculus classes, but rather by a precomputed best fit polynomial.

Representation 1 : Polynomials

Smooth functions on a finite interval can be well approximated by polynomials. In the complex plane, a Taylor series

$$f(x) = \sum_{k \geq 0} a_k (x - c)^k \tag{8}$$

converges inside a disk that reaches to the first singularity. The first p terms form a truncated series that will be highly accurate near c:

$$f(x) = \sum_{0 \leq k \leq p-1} a_k (x - c)^k \tag{9}$$

Alternatively, one can choose p interpolation points obtaining a $p - 1$ degree polynomial that are exact at least at the p points. Both choices are polynomials. We can think of the Taylor series as the limit of choosing all p point to "coalesce" to the identical value "c".

Representation 2 : Multipole Series

The typical form of a multipole series is

$$f(x) = \sum_{k \geq 1} a_k / (x - c)^k. \tag{10}$$

Such a series converges outside a disk. The first p terms will be most accurate far from the disk centered at c. Again one can choose p interpolation points instead and obtain a slightly different expansion of

$$f(x) = \sum_{1 \leq k \leq p} a_k / (x - c)^k. \tag{11}$$

Representation 3 : Virtual Charges

If we choose $x_1, \ldots, x_p$ in advance, we can consider functions of the form

$$\sum_{1 \leq k \leq p} a_k / (x - x_k). \tag{12}$$

Such functions form the vector space of functions with poles at only the x_i's. One can interpolate such a function at p points y_i to compute the a_i.

Alternatively, we need not preselect the positions $x_1, \ldots, x_p$. In this case the space of functions with p poles is not a vector space, though the infinite space of functions with finitely many poles is. This creates interesting challenges and opportunities.

3.2 Rounding Choices

As we introduced above in Section 3.1, we can choose a function space, but still many choices remain of how to represent an infinite function.

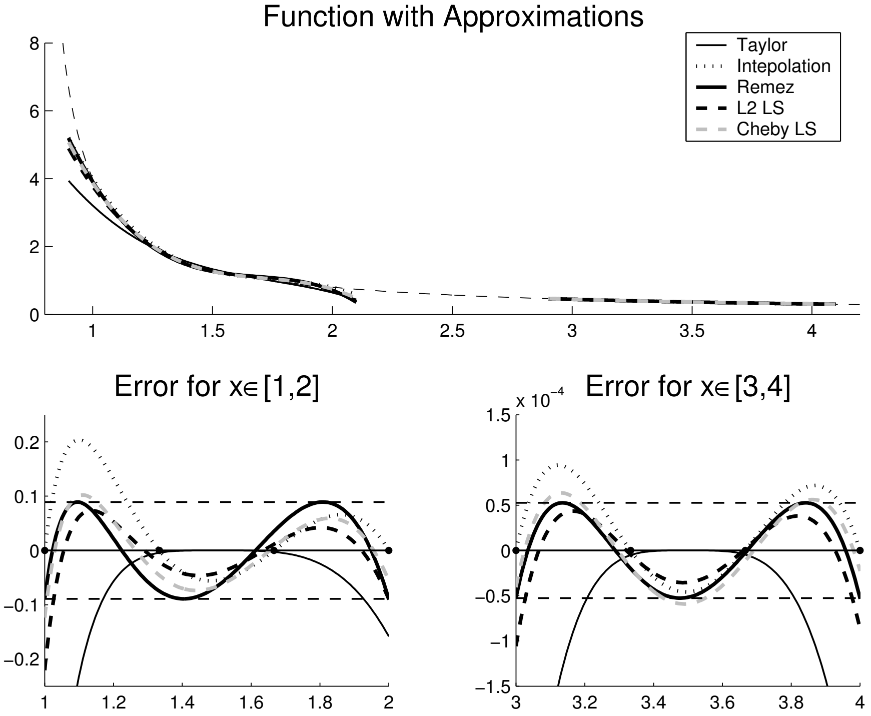


Figure 4: Rounding choices for approximation by cubic polynomials ($p = 4$). The intervals $[1, 2]$ and $[3, 4]$ are approximated separately.

3.2.1 Polynomial Approximation

Consider for example the function $1/(x - 0.75)$ in Figure 4 on the interval $[1, 2]$ and also the interval $[3, 4]$. We seek separately the best cubic polynomials that approximate this function on these intervals.

First thing to notice is that $f(x) = 1/(x-0.75)$ has a better approximation at $[3, 4]$ then $[1, 2]$. In both cases the Taylor series at the center of the interval is among the worst approximations. Interpolation gives a better approximation. Better yet are the "L_2 least squares" approximation which minimizes the integral of the squared error, and the "L_∞ Remez" approximation which minimizes the absolute error.

We summarize the choices below:

Interpolation Given any vector space of functions, the problem of interpolation becomes a simple problem of solving a linear system of equations. Issues of numerical conditioning may arise in practice.

Typical choices of Interpolation Points for polynomials are

a. All points coincide (Taylor Series)

b. Equally spaced points

c. Chebychev Points (more points closer to the exterior)

d. Adaptive Choice (more points where function varies the most)

Approximation Approximation means we do not primarily seek the function to be accurate at specific points, but rather we seek a small error. In general this is an optimization problem though special algorithms are often available.

Typical Approximation Schemes include

a. L_2 norm (least squares). This reduces to an ordinary linear least squares problem

$$\min_{\{g(x)\}} \int (g(x) - f(x))^2. \tag{13}$$

b. L_∞ norm

$$\min_{\{g(x)\}} \max_x |g(x) - f(x)|. \tag{14}$$

For polynomials, the Remez algorithm may be used to get the best fit. The error equioscillates between two horizontal lines as is clear in Figure 4. Remez is an interesting algorithm to implement but may be too expensive for multipole purposes.

The two above choices represent minimization of norms of the absolute error. We are not aware of much work in the direction of minimizing the relative error: $\frac{|f(x)-g(x)|}{|f(x)|}$. We believe this is an opportunity for future research.

3.2.2 Multipole Approximation

A multipole approximation is really a polynomial approximation at ∞. Simply by plugging in $y \to 1/(x-c)$, this can be readily seen. Figure 5 shows the difference between a multipole approximation and a polynomial approximation (Taylor Series) for the same function.

3.2.3 Virtual Charges

One can use most of the techniques above to approximate using virtual charges. A technique that we have not seen in the literature that we wish to discuss is the use of the Lanczos algorithm to pick p positions and p charges when the approximation function consists of N positions and N positive charges with N presumed $\gg p$.

Philosophically the use of Lanczos is an interesting choice. To us it says that the next best thing in linear algebra to a linear problem is an eigenvalue problem. Though nonlinear, tridiagonal eigenvalues problems are familiar with rock solid software available (see http://math.mit.edu/~persson/mltrid for a MATLAB version).

The idea is to begin with a diagonal matrix D that contains the original positions, and a vector $\boldsymbol{q}$ whose square represents the charges normalized to have sum 1. We then project D to a Krylov space $[\boldsymbol{q} \;\; D\boldsymbol{q} \;\; \ldots \;\; D^{p-1}\boldsymbol{q}]$ using the Lanczos method obtaining a $p \times p$ tridiagonal matrix T whose eigenvalues are the new p positions, and the square of the first elements of each eigenvector are the new charges.

Our Lanczos method is summarized in Algorithm 1:

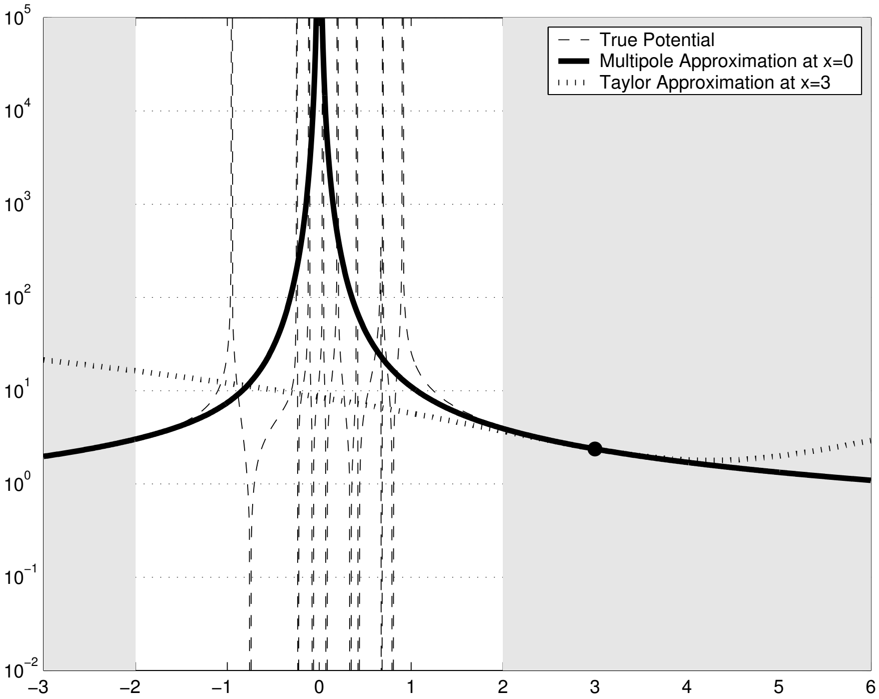


Figure 5: Approximation by multipole and Taylor expansions, both with $p = 3$ terms.

Algorithm 1: Lanczos Virtual Charge Approximation

Description: Approximate n charges $\boldsymbol{x}, \boldsymbol{q}$ by p virtual charges $\boldsymbol{x}_{\text{vc}}, \boldsymbol{q}_{\text{vc}}$
Input: $\boldsymbol{x}, \boldsymbol{q} \in \mathbb{R}^n, \text{sum}(\boldsymbol{q}) = 1$, integer p
Output: $\boldsymbol{x}_{\text{vc}}, \boldsymbol{q}_{\text{vc}} \in \mathbb{R}^p$

```
$D = \text{diag}(\boldsymbol{x})$                                   Charge positions on diagonal
$\boldsymbol{q}_{\text{old}} = \boldsymbol{0}$
$b_0 = 0$
for $k = 1$ to $p$                                                   Lanczos iterations
    $a_k = \boldsymbol{q}^T D\boldsymbol{q}$                          Note that $\boldsymbol{q}^T D\boldsymbol{q} = \sum x_i q_i^2$
    $\boldsymbol{q}_{\text{new}} = D\boldsymbol{q} - a_k\boldsymbol{q} - b_{k-1}\boldsymbol{q}_{\text{old}}$    Again a trivial matrix operation
    $b_k = \|\boldsymbol{q}_{\text{new}}\|$
    $\boldsymbol{q}_{\text{new}} = \boldsymbol{q}_{\text{new}}/b_k$
    $\boldsymbol{q}_{\text{old}} = \boldsymbol{q}$, $\boldsymbol{q} = \boldsymbol{q}_{\text{new}}$
end
```

$$T = \begin{pmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{p-1} \\ & & & b_{p-1} & a_p \end{pmatrix} \qquad \text{Form symmetric tridiagonal matrix}$$

```
$TV = \Lambda V$                                                     Eigenvalue decomposition of $T$
$\boldsymbol{x}_{\text{vc}} = \text{diag}(\Lambda)$                  Vector of eigenvalues
$\boldsymbol{q}_{\text{vc}} = (V_{11}^2, V_{12}^2, \ldots, V_{1p}^2)$    First component of each eigenvector
```

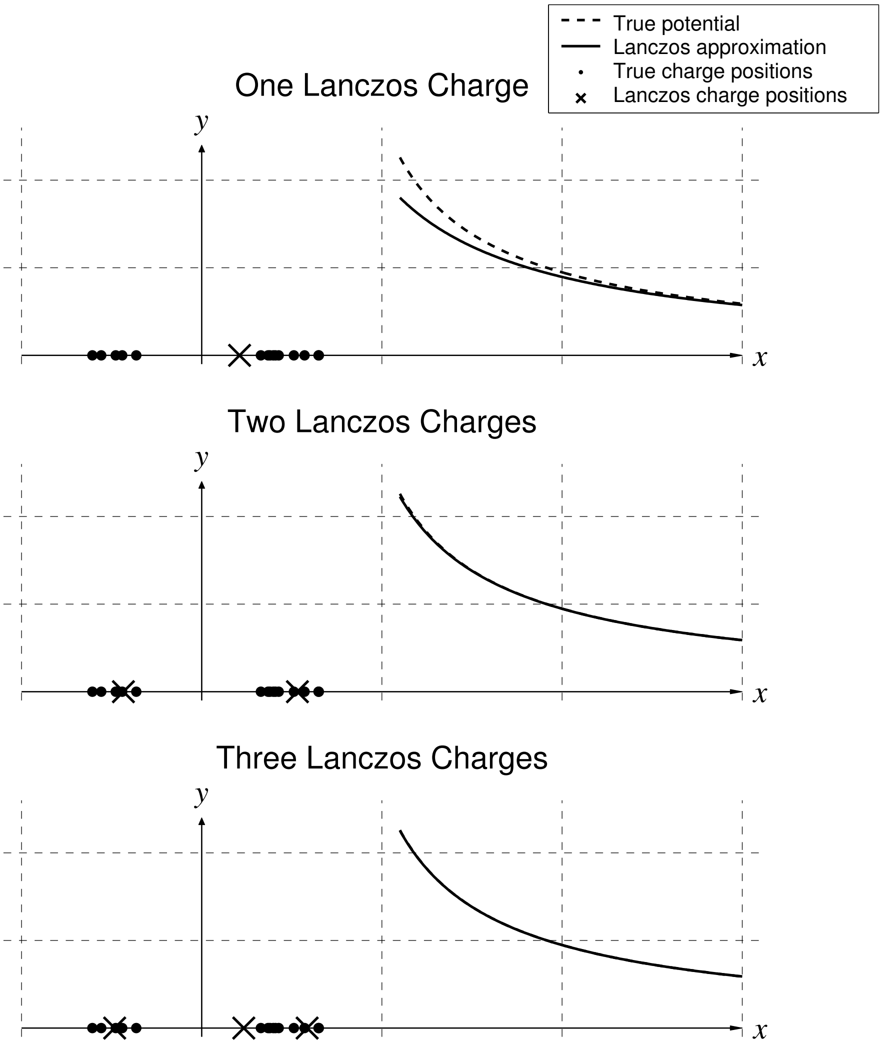


Figure 6: Lanczos approximation with $p = 1, 2, 3$ terms. For $p = 1$ the result is the standard "center-of-mass" approximation.

A demonstration of the algorithm is shown in Figure 6. When $p = 1$ we obtain the sum of the masses concentrated at the center of mass. This is equivalent to $p = 2$ in the multipole algorithm. In general, through the magic of Gaussian quadrature and orthogonal polynomial theory, we obtain p virtual charges and positions that is equivalent to $2p$ terms in a multipole expansion.

The parameter count is 1 normalization condition + $(p - 1)$ parameters for the charges + p

positions $\equiv$ 1 normalization condition+ $(2p-1)$ multipole terms.

We recognize that many readers interested in FMM may not be as familiar with the Lanczos method. Lanczos is often considered to be an eigenvalue method – this application to solve the nonlinear problem of where to place virtual charges is a complete surprise! That we get double the accuracy is a consequence of another classical piece of mathematics – Gauss Quadrature. The connections are surprising and wonderful.

3.3 Adding Functions

3.3.1 Shifting and Flipping

Once functions are represented on a computer we must think carefully how we can add them. Two issues arise.

Issue 1: An algebraic issue

Typically our two functions will not have exactly the same representation. For example, they may have the form (8) , but with two different centers $c = c_1$ and $c = c_2$. Then an algebraic manipulation to a common center is necessary before we can add. Consider for example adding

$$1 + (x-1) + 3(x-1)^2 \quad \text{to} \quad 4 + (x+3) + 5(x+3)^2. \tag{15}$$

When there are p coefficients, typically, a $p \times p$ matrix expresses this "shift". If we are adding multipole series, the same issue arises.

Issue 2: An analytic issue

When going up the tree in the hierarchical algorithms in Section 2.2 we typically have some sort of multipole series at the leaves of the tree. When we go down the tree, we typically have a Taylor series. This "flip" is essential because on the way up, we are expressing functions that we need to be accurate outside the interval, but on the way down, the functions need to be accurate inside the interval. Recall that analytically, Taylor series converge inside a disc and after truncation they are accurate inside a smaller disc. Conversely, analytical multipole series converge outside a disc and truncated ones are accurate outside a larger disc.

3.3.2 The Pascal Matrix: Standard shifts and flips in 1-D

Binomial Identities

$$\begin{array}{rcll} (1-x)^{-(j+1)} & = & \sum_{i=0}^{\infty} \binom{i+j}{i} x^i & \leftarrow P \\ (1+x)^i & = & \sum_{j=0}^{i} \binom{i}{j} x^j & \leftarrow L \\ (x-1)^{-(i+1)} & = & \sum_{j=i}^{\infty} \binom{j}{i} x^{-(j+1)} & \leftarrow L^T \end{array}$$

The standard Taylor series and multipole series choices require flipping and shifting. This is closely related to the "Pascal Matrix" $P_{ij} = \binom{i+j}{i}$ for $i, j = 0, \ldots, \infty$. The P corresponds to a flip, the L to a Taylor shift, and L^T to a multipole shift. Indeed these cases are unit cases, i.e., are shift by one and the flip also moves the center by one.

The proper scaling for non unit shifts is given below:

"Flipping" (multipole → taylor)

$$\begin{aligned} w &= \text{FLIP } v \\ \sum_{i=0} w_i(x-d)^i &= \sum_{j=0} v_j/(x-c)^{j+1} \end{aligned}$$

$$\text{FLIP}_{ij} = (c-d)^{-i} P_{ij} (d-c)^{-(j+1)}$$

Proof: Differentiate i times then evaluate at $x = d$ or cleverly use $(1-x)^{-(j+1)} = \sum \binom{i+j}{i} x^i$.

"Shifting" (taylor → taylor)

$$\begin{aligned} w &= \text{T-SHIFT } v \\ \sum_{i=0} w_i(x-d)^i &= \sum_{j=0} v_j(x-c)^j \end{aligned}$$

$$\text{T-SHIFT}_{ij} = (d-c)^{-i} L_{ij} (d-c)^j$$

"Shifting" (multipole → multipole)

$$\begin{aligned} w &= \text{M-SHIFT } v \\ \sum_{i=0} w_i/(x-d)^{i+1} &= \sum_{j=0} v_j/(x-c)^{j+1} \end{aligned}$$

$$\text{M-SHIFT}_{ij} = (c-d)^{i+1} L^T_{ij} (c-d)^{-(j+1)}$$

There are many interesting ways to generalize the above. Even in one dimension it is fantastic that we can write generalized formulas for orthogonal polynomials:

$$\begin{aligned} m_i(x) &= \textstyle\sum_{j=0}^{\infty} m_{i+j} x^j & \leftarrow H \\ \pi_i(x) &= \textstyle\sum_{j=0}^{i} < \pi_i, x^j > x^j & \leftarrow L \\ f_i(x) &= \textstyle\sum_{j=i}^{\infty} < x^i, \pi_j > x^{-j+1} & \leftarrow L^T, \end{aligned}$$

where $w(x)$ is any weight function, $\{\pi_j(x)\}$ the corresponding orthogonal polynomials, and $< f, g >$ denotes the inner product $\int f(t)g(t)w(t)$, and $f_i(x) = \int \frac{\pi_i(t)w(t)}{x-t}\, dt$. Ultimately $LL^T = H$, where $L_{ij} =< \pi_i, x_j >$ and $H_{ij} = \int x^{i+j} w(x)\, dx$. There are nice relationships between all of these functions and the associated orthogonal polynomials to the π_j...

This generalizes to higher dimensions and many interesting mathematical situations.

4 Experiments with Direct FMM

In the previous sections we have described how the FMM consists of two parts – an algorithm for fast addition with exclusion and approximate representation of functions. We also showed how a simple direct algorithm can be used instead of the hierarchical method, at least for the "analytical case" of scalar numbers. The natural question is now: Can our simpler direct algorithm be used when the x_i are replaced with finite function representations? Returning to the 1-D problem, if we use ordinary multipole expansions for the cumulative sum y_i^{p} from (6) we get the problem that we have to evaluate the expansion for $r < 2R$. When we evaluate the potential at the point locations in cell number i, we use the finite function representation y_{i-2}^{p}. Since this corresponds to all particles in the cells $1, \ldots, i-2$, we do not have the "factor of 2" between the particles and the evaluation points.

One solution would be to use an alternative representation of the potentials. Unlike multipole expansions, which are accurate outside the relative region of two radii, our new representation must be accurate outside the absolute region of one cell width. We will evaluate the approximate function at a relative distance $1+\epsilon$, where ϵ gets smaller as i gets larger. However, in absolute terms there is always one cell width between the particles and the evaluation points.

We have made some initial experiments using virtual charges as function representation. For cells $i = 1, \ldots, i$, we choose p fixed locations x_i^{vc} and approximate the field by the potential:

$$y(x) \approx \sum_{i=1}^{p} \frac{q_i}{|x - x_i^{\mathrm{vc}}|} \tag{16}$$

The charges q_i are determined by least-square fitting of the original potential. Our 1-D experiments show that accurate results are obtained when the density of virtual charges (the locations x_i^{vc}) is concentrated close to the right end (that is, close to the evaluation points). The algorithm for the far field is then (in pseudo-code):

```
$x_{\mathrm{old}}^{\mathrm{vc}}, q_{\mathrm{old}}^{\mathrm{vc}}$ = empty
for $i = 1 : n-2$
  Choose $p$ new v.c. positions $x_{\mathrm{new}}^{\mathrm{vc}}$
  Compute $q_{\mathrm{new}}^{\mathrm{vc}}$ by fitting to charges in cell $i$ and v.c.'s $x_{\mathrm{old}}^{\mathrm{vc}}, q_{\mathrm{old}}^{\mathrm{vc}}$
  Evaluate new v.c. $x_{\mathrm{new}}^{\mathrm{vc}}, q_{\mathrm{new}}^{\mathrm{vc}}$ at locations in cell $i+2$
  $x_{\mathrm{old}}^{\mathrm{vc}} = x_{\mathrm{new}}^{\mathrm{vc}}$, $q_{\mathrm{old}}^{\mathrm{vc}} = q_{\mathrm{new}}^{\mathrm{vc}}$
end for
```

and similar for the suffix part. Below are some observations:

- We use only one type of approximation (virtual charges), compared to the two representations in the original FMM (multipoles and Taylor series).
- We evaluate the approximations as soon as they are available, and consequently we only store one function representation in memory.
- During the first iterations, the evaluation points are well separated from the initial and virtual charge locations (when $i = 1$ we have the same factor of 2 as in the FMM). For higher i values, the absolute separation is still one cell, but the relative separation

decreases. In our implementation we observed this in that we required higher values of p (a guess is that it increases as the logarithm of the number of cells). It is not clear if this is compensated by the fact that we do not have to store all the representations, and the total computational cost will depend on the algorithm for computing new approximations.

- The algorithm is remarkably simple to describe and to implement. Most of the complexity is in the computation of the function approximations.

- The number of cells n does not have to be a power of 2.

Our main goals for the future development of the algorithm are to find more strict results regarding accuracy and computational cost, and to extend the algorithm to higher dimensions.

References

[1] J. Barnes and P. Hut. A hierarchical o(nlogn) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[2] R. Beatson and L. Greengard. A short course on fast multipole methods. In M. Ainsworth, J. Levesley, W. Light, and M. Marletta, editors, *Wavelets, Multilevel Methods and Elliptic PDEs*, pages 1–37. Oxford University Press, 1997.

[3] H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *J. Comput. Phys.*, 155(2):468–498, 1999.

[4] A. Dutt, M. Gu, and V. Rokhlin. Fast algorithms for polynomial interpolation, integration, and differentiation. *SIAM J. Numer. Anal.*, 33(5):1689–1711, 1996.

[5] A. Edelman and G. Strang. Pascal matrices. *American Math. Monthly*, 111:189–197, 2004.

[6] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[7] L. Greengard and V. Rokhlin. A new version of the fast multipole method for the laplace equation in three dimensions. *Acta Numerica*, pages 229–269, 1997.