Fast Multipole: It's all about Adding Functions

Just a start for now

March 18, 2002

I must be honest. It has taken me a long time to fully understand the famous Fast Multipole Method (FMM). Traditionally, FMM is portrayed as a fast matrix times vector, a useful point of view that I will use too, but for me the essence of FMM is about accurately adding functions with finite storage.

Can we set the stage for a moment with the example of finite precision addition? We begin with the mathematical notion of addition of real numbers. We all know what this means. On a computer, real numbers are rounded to d-bit floating point precision with a loss known as "rounding error." Floats are "added" in a manner whose full specification is often left to chip designers, but in any event rounding errors are associated with the addition as well.

As children we all have learned about adding numbers, and by high school we advance to the notion of adding functions. We learn that $\sin^2(x) + \cos^2(x) = 1$ and $\log x + \log x = \log x^2$. On a computer, FMM represents functions by p coefficients with a loss known as "approximation error." Addition of these representations is performed in a manner known to FMM experts, and in any event are associated with further approximation error.

Even those familiar with FMM may find this approach intriguing. The functions that we add in FMM might look something like $f(x) = q/(x-c)$ or $f(x) = q\log(x-c)$ or even $f(x) = \cot(x-c)$. The main point is that there is a singularity ($x = c$), and without being very precise, "slow motion" away from the singularity. $f(x) = \exp(-(x-c)^2)$ would not be a candidate for FMM.

1 Exclusive adds and other recursive algorithms

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

$$y = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} x.$$

The exclusive add is the essential operation of FMM, though FMM usually implements a "doubly exclusive" add which excludes each item and its nearest neighbors. We will say more about this later.

How should we implement such 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.

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)

The simple MATLAB function implements this

```
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
```

Another interesting example shows that if the inputs are scaled very differently, the recursive 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:

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

Our final example shows that the same function works on symbolic functions:

```
>> 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)]
```

In Matrix notation, the recursive algorithm is based on a decomposition that repsents a rank n matrix as the sum of a rank $n/2$ matrix and a block diagonal matrix. We use dots to represent 0's:

$$\begin{pmatrix} . & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & . & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & . & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & . & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & . & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & . & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & . & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & . \end{pmatrix} = \begin{pmatrix} 1 & . & . & . \\ 1 & . & . & . \\ . & 1 & . & . \\ . & 1 & . & . \\ . & . & 1 & . \\ . & . & 1 & . \\ . & . & . & 1 \\ . & . & . & 1 \end{pmatrix} \begin{pmatrix} . & 1 & 1 & 1 \\ 1 & . & 1 & 1 \\ 1 & 1 & . & 1 \\ 1 & 1 & 1 & . \end{pmatrix} \begin{pmatrix} 1 & . & . & . \\ 1 & . & . & . \\ . & 1 & . & . \\ . & 1 & . & . \\ . & . & 1 & . \\ . & . & 1 & . \\ . & . & . & 1 \\ . & . & . & 1 \end{pmatrix}^T + \begin{pmatrix} . & 1 & . & . & . & . & . & . \\ 1 & . & . & . & . & . & . & . \\ . & . & . & 1 & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & . & 1 \\ . & . & . & . & . & . & 1 & . \end{pmatrix}.$$

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

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

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.)

The most famous case is that of prefix: $(y_i = \sum_{i \le j} x_j.)$ The matrix is the lower triangular matrix of ones.

```
function y=prefix(x)

n=length(x);
if n==1, y=x; else
  w=x(1:2:n)+x(2:2:n);                           % Pairwise adds
  w=prefix(w);                                   % Recur
  y(1:2:n)= x(1:2:n)+[0 w(1:end-1) ]; y(2:2:n)=w; % Update Adds
end
```

The reader can write out the matrix decomposition. Parallel prefix is justly famous in this author's opinion because so many apparently serial algorithms can be parallelized with this operation.

We return now to FMM. The one dimensional FMM computes $y = M_8 x$, where x is a vector of functions, and

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

If M_8 is thought of as a block matrix with 2×2 blocks, then the superdiagonal and subdiagonal blocks are

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

respectively. This requires three updates by x needed for each element, with the usual asymmetry between left and right:

code to be added

2 Finite Representations of Functions

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?

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

$$f(x) = \sum a_k (x-c)^k.$$

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

The typical form of a multipole series is

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

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 expanison of

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

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 by the Taylor series seen in calculus classes, but rather by a precomputed best fit polynomial. This is practical the most widely used functions.

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

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

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. This approximation method can be quite accurate for a function that has a large number of poles contained in an interval I and the function is to be well approximated far from I.

Given a large (or even infinite) dimensional function space, we can use the svd to compress to a lower dimensional function space. (More on this later...............)

*** Graphs of truncation and interpolation perhaps

3 Adding Functions

4 Beyond One Dimension

5 Below the tree