ABSTRACT

Title of thesis: IMPROVING RADIAL BASIS FUNCTION INTERPOLATION VIA RANDOM SVD PRECONDITIONERS AND FAST MULTIPOLE METHODS

Kerry Cheng, Master of Science, 2013

Thesis directed by: Professor Ramani Duraiswami
Department of Computer Science

Recent research in fast-multipole algorithms has yielded approximation algorithms that compute specific matrix vector products to any specified accuracy in linear time. This thesis tries to combine this with recent research in randomized algorithms that has developed fast ways to compute rank-k SVDs of an $M \times N$ matrix. This combination yields an approximate SVD in $O(k \max(M, N))$ time. We demonstrate this and explore its use in developing a novel scattered-data interpolation algorithm in three dimensions. The use of sinc functions in three dimensions is also explored, specifically their ability to accurately interpolate some standard functions. We find that the width parameter plays an important role, and suggest a prescription for its selection. As with other RBF interpolation algorithms, interpolating N points requires the solution of a dense linear system, which has $O(N^3)$ cost. We explore two uses of the fast randomized SVD to reduce this cost. First, we use the randomized SVD to come up with a solution to the linear system. Next, we use a preconditioned Krylov iterative method (GMRES) with a low rank SVD as a preconditioner. Results are presented, and the method is found promising.

IMPROVING RADIAL BASIS FUNCTION INTERPOLATION VIA RANDOM SVD PRECONDITIONERS AND FAST MULTIPOLE METHODS

Kerry Cheng

August 13, 2013

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2013

Advisory Committee:
Professor Ramani Duraiswami, Chair
Professor Larry Davis
Professor David Mount

Acknowledgments

Many thanks to Dr. Ramani Duraiswami, my advisor, for his guidance, support, patience, encouragement, and significant suggestions throughout my undergraduate independent study and graduate research to complete this thesis. I would like to thank Dr. Nail Gumerov as well, for his guidance and advice.

Thanks to Professor Jeff Foster, Associate Chair for Graduate Studies, and Graduate School of University of Maryland, for accepting me as a graduate student while I was still in my junior year.

Last but not least, I would like to thank my parents for all their love and support.

List of Tables

3.1 SVD Results: $N = 500$; $k = 200, l = 220$ for Random SVD 7
3.2 SVD Results: $N = 2000$; $k = 500, l = 520$ for Random SVD 8
3.3 SVD Results: $N = 5000$, $k = 1000, l = 1020$ for Random SVD . . . 8
3.4 SVD Results: $N = 10000$, $k = 2500, l = 2520$ for Random SVD . . . 8
4.1 Comparing Wavenumber: $N = 500$ 12
4.2 Comparing Wavenumber: $N = 1000$ 12
5.1 FMM in GMRES: $N = 500$ 14
5.2 FMM in GMRES: $N = 2000$ 14
5.3 FMM in GMRES: $N = 3000$ 14
5.4 FMM in GMRES: $N = 5000$ 14
5.5 Random SVD: $N = 500$ 15
5.6 Random SVD: $N = 2000$ 16

List of Figures

3.1 Singular Values for $N = 500$ 7
3.2 Time vs. N 8
4.1 Sinc Function 10
4.2 Normal k 11
4.3 Large vs. Small k 11
4.4 Interpolation Error vs. a 12
5.1 Using FMM as a matrix-vector in GMRES 15

Contents

1 Introduction **1**
- 1.1 Fast Multipole Method . . . 1
  - 1.1.1 Matrix Vector Product . . . 1
- 1.2 Singular Value Decomposition (SVD) . . . 1
  - 1.2.1 Randomized Algorithm . . . 2
    - Algorithm . . . 2
- 1.3 Sinc Function . . . 2
- 1.4 Radial Basis Function Interpolation . . . 3
  - 1.4.1 Setting Up the System . . . 3
  - 1.4.2 Solving the System . . . 4

2 Methods **5**
- 2.1 FMM . . . 5
- 2.2 Code . . . 5
- 2.3 Testing . . . 5

3 FMM Accelerated Rank k SVD **7**
- 3.1 Implementation . . . 7
- 3.2 Results . . . 7
  - 3.2.1 Singular Values . . . 7
  - 3.2.2 Error . . . 7
  - 3.2.3 Time . . . 8

4 Radial Basis Function Interpolation with the sinc function **10**
- 4.1 k Parameter . . . 10
- 4.2 Results . . . 11

5 Iterative Methods and Preconditioning **13**
- 5.1 Background . . . 13
  - 5.1.1 Krylov Subspace Methods . . . 13
  - 5.1.2 Preconditioning . . . 13
- 5.2 Results . . . 14
  - 5.2.1 GMRES . . . 14
  - 5.2.2 Preconditioning . . . 15

6 Conclusions **17**
- 6.1 Future Directions . . . 17

7 Appendices **18**
- 7.1 Source Code . . . 18
  - 7.1.1 FMM `mexfunction` . . . 18
  - 7.1.2 Random SVD MATLAB Source Code . . . 20
  - 7.1.3 Random SVD using FMM MATLAB Source Code . . . 20
- 7.2 Thesis Presentation . . . 21

1 Introduction

1.1 Fast Multipole Method

The Fast Multipole Method (FMM) was developed by Greengard and Rokhlin in 1987. The algorithm could rapidly evaluate potential and force fields for systems with a large number of particles. Previously, such calculations required work on the order of $O(N^2)$ to evaluate all pairwise interaction, unless a truncation method was used. However, the FMM algorithm could perform the same calculations to arbitrary accuracy in the order of $O(N)$ time [4].

1.1.1 Matrix Vector Product

While the FMM was initially developed to accelerate the calculation of forces in the n-body problem, it has since been used to improve many other algorithms. In particular, it can speed up sums of the following type:

$$s(x_j) = \sum_{i=1}^{N} \alpha_i \phi(||x_j - x_i||)$$

[5]. This is exactly the sort of sum that is calculated when performing a matrix-vector product with a matrix A where $A_{ij} = \phi(||x_i - x_j||)$. For an arbitrary vector x,

$$[Ax](j) = \sum_{i=1}^{N} x_i \phi(||x_j - x_i||)$$

Therefore, an FMM routine can be used to dramatically speed up any matrix vector products of this type.

1.2 Singular Value Decomposition (SVD)

The Singular Value Decomposition (SVD) of an $N \times N$ matrix A is defined as follows:

$$A = U\Sigma V^T$$

where U and V are $N \times N$ matrices whose columns are orthonormal and Σ is a diagonal matrix containing the N singular values on the diagonal. Using this, it is easy to form a pseudoinverse:

$$A^{-1} = V\Sigma^{+}U^T,$$

where Σ^{+} is the psuedoinverse of Σ. Σ^{+} can be formed by replacing every non-zero element on the diagonal with its reciprocal. Thus, given a linear system $Ax = b$, it can be solved by finding the SVD and multiplying.

$$A^{-1}Ax = A^{-1}b$$
$$x = A^{-1}b$$

The standard algorithm for calculating the SVD of a matrix involves two steps [10]. First, the matrix needs to be reduced to a bidiagonal one. This can be done in $O(mn^2)$ operations for an $M \times N$ matrix, where $M \geq N$. Then the SVD of this bidiagonal matrix is found via an iterative algorithm. This step is usually computed to some constant precision and is less computationally expensive than the first step. Therefore, the overall cost is $O(mn^2)$ [10].

1.2.1 Randomized Algorithm

A randomized algorithm was developed by Martinsson et al., which constructs a rank-k approximation of the SVD using l random vectors ($l > k$) [7]. When holding $l - k$ to be 20, the algorithm's accuracy was found to be of roughly the same order as the best possible rank-k approximation. Additionally, the algorithm's failure rate was almost negligible (10^{-17} was typical). This efficiency and accuracy suggest it could be used as an effective preconditioner.

Algorithm For an arbitrary $N \times N$ matrix A, it is approximated by a matrix Z of rank k using l

1. The algorithm approximates an $N \times N$ matrix A by first applying A^{T} to the collection of l random vectors of length N. This gives a rectangular matrix R of size $N \times l$.

2. Using an SVD of $R = U'\Sigma'V'^{\mathrm{T}}$, let Q be the first k columns of U. This will be a matrix of dimensions $N \times k$ and its columns are a good approximation of an orthonormal basis of the range of A [7].

3. Then, the $N \times k$ matrix is calculated, $T = AQ$, requiring k matrix vector products.

4. Now another SVD is formed: $T = U\Sigma W^{\mathrm{T}}$, where T is a $k \times k$ matrix.

5. Compute a final matrix vector product $V = QW$. U, Σ, V will form an SVD of the original matrix A, such that, $A = U\Sigma V^{\mathrm{T}}$.

Steps 1 and 3 both require multiple matrix vector products between A and arbitrary vectors. Therefore, these operations could also be replaced with an FMM routine to improve speed and convergence.

This scheme is efficient whenever A and A^T can be applied rapidly to arbitrary vectors [7]. Thus, it is clear that the FMM could be useful in improving this randomized algorithm.

1.3 Sinc Function

The sinc function is one that has been used in a lot of digital sample processing. It is defined as:

$$\mathrm{sinc}(x) = \frac{\sin(x)}{x}, \qquad \text{where } x \in \mathbb{R}$$

The radial basis version, applicable for any number of dimensions, is:

$$\text{sinc}(x) = \frac{\sin(k||x - x_*||)}{k||x - x_*||}, \qquad \text{where } x \in \mathbb{R}^d$$

As a radial basis function, this is centered on a specific point. And its value at any point in $\mathbb{R}^d$ is completely dependent on the distance to that center.

1.4 Radial Basis Function Interpolation

As mentioned previously, radial basis functions are the set of functions whose values at a certain point are based on the distance from some center point. While these functions are simple, they have found significant use as one of the primary methods of interpolating multidimensional scattered data. The Radial Basis Function (RBF) method is based on approximating multivariable functions as linear combinations of terms based on a function of one variable (the radial basis function). As mentioned previously, functions which are either not known or too complex to evaluate at many points in its domain can then be approximated at those points with a much simpler function. This allows the approximated function to be evaluated much more efficiently and quickly.

In many domains, functions that need to be approximated are of many variables, such as in neural network learning. Radial basis functions are commonly used, due to their efficiency and accuracy. An important property of RBF interpolations is the data dependence of the approximation space. As this method can be used for functions of any dimensionality, it has wide applicability. In addition, there are no restrictions placed on the data, other than being at distinct points. Many methods such as multivariate spline methods or the Finite Element method depend on the data having a specific geometry. The sampled points can be irregularly placed rather than necessarily on a uniform grid [3]. In practical applications, this can be difficult, if not impossible. Finally, RBF interpolants have been found to be very accurate, even when the number of interpolation points is small [3].

Currently, the multiquadric, Gaussian, inverse multiquadric, and thin plate spline functions are the most studied and used in interpolations [3]. However, other functions could demonstrate good performance, especially in specific applications. It is our hope that the sinc function can also show promise.

1.4.1 Setting Up the System

Assume that data values are given at N points in d dimensional space. We want our interpolant to be a linear combination of the N RBF functions centered at these N points. It will be of the form:

$$f(x) = \sum_{j=1}^{N} \lambda_j \cdot \phi(||x - x_j||) \tag{1.1}$$

Since the interpolation must fit at all N points, we have a linear system of N equations and N variables. For each point x_j,

$$f(x_j) = \lambda_1 \cdot \phi(||x_j - x_1||) + \cdots + \lambda_N \cdot \phi(||x_j - x_N||)$$

These can be represented in matrix form ($Ax = b$):

$$\begin{bmatrix} \phi(||x_1 - x_1||) & \cdots & \phi(||x_1 - x_N||) \\ \vdots & \ddots & \vdots \\ \phi(||x_N - x_1||) & \cdots & \phi(||x_n - x_N||) \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_N \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_N \end{bmatrix} \tag{1.2}$$

Note that A is positive definite.

1.4.2 Solving the System

This system can be solved with a wide variety of methods. Simple direct ones, like Gaussian elimination, could be used to obtain an exact solution. However, in practical applications, these systems are far too large to be solved efficiently in this way. Thus, better ways of solving these systems must be used.

2 Methods

All testing was done in a combination of MATLAB and FORTRAN. Most of the code was written in MATLAB, with calls to FORTRAN routines for FMM calculations.

To simplify testing, all interpolation was done in the unit cube in 3 dimensions, i.e., $[0-1] \times [0-1] \times [0-1] \subset \mathbb{R} \times \mathbb{R} \times \mathbb{R}$. For a given number N, the function is interpolated at N random locations. The matrix A can be calculated by looping through every combination of points i and j, and finding $\text{sinc}(k * ||\text{points}_i - \text{points}_j||)$. This creates the system to be solved, $AX = F$ as mentioned in (1.2). The system can be solved for X via various methods. This includes the SVD as mentioned earlier or an iterative method like GMRES [1].

To test the accuracy of this interpolant, a 3-dimensional mesh is created on the domain with uniform distance between mesh points. At each mesh point, the approximation is calculated as described in (1.1), with X corresponding to λ. This is compared to the exact value of the interpolated function to find the error. After processing each point in the mesh, the L_2 and L_{inf} norms are calculated for both the absolute and relative errors.

2.1 FMM

The FORTRAN FMM routine used was developed by Duraiswami and Gumerov to accelerate the iterative solution of the boundary element method for the three-dimensional Helmholtz equation [6]. As the points are not moving, the sources and receivers of the FMM routine are both simply set to the N random points. The wavenumber of the FMM is set to the k parameter to ensure they are using the same scaling.

2.2 Code

In order to test the FMM, it was necessary to call the FORTRAN FMM routine from a MATLAB function.

Due to limitations of MATLAB, it wasn't possible to pass arrays of structures. Due to the way complex numbers are stored, this made it impossible to natively call the FORTRAN routine using just `libpointer` [2]. Instead, a FORTRAN `mexfunction` had to be created to serve as a bridge between the MATLAB and FORTRAN data.

It takes in arrays for native MATLAB input and output arrays. From this, it can dynamically allocate memory for FORTRAN data and appropriately set them. Then, the FMM routine can be easily called from the shared library. The final matrix vector product returned is then converted into a MATLAB pointer and passed back. All this allows MATLAB to act as if the FORTRAN routine is a native MATLAB function. The source code for this function can be found in section 7.1.1.

2.3 Testing

Tests were done to compare the accuracy and speed of these methods. A standard convection reaction was used as the problem to be interpolated. Specifically, the

convection-diffusion reaction, defined as:

$$T(x, y, z) = e^{-x} + e^{-y} + e^{-z} \tag{2.1}$$

[8].

Since it is possible to find interpolants that are too narrow, the mesh needed to be more dense than N. This ensures the accuracy measurement won't miss areas of poor accuracy between the sampled points. As such, a mesh distance of .01 was used, giving 101 mesh points along each dimension. This corresponds to $101^3 = 1030301$ total mesh points. For the values of N tested, this was more than sufficient.

Multiple trials of each test were done to find an average. For smaller values of N, 20 trials were used, but this became infeasible for larger ones. Therefore, for smaller values of N, only 5 trials were used.

When evaluating the method on the mesh, both the absolute and relative error are recorded. After finishing the L_2 and L_{inf} norms of both are given. The L_2 norms are divided by N to normalize them. Additionally, the number of iterations required for convergence and time taken (in seconds) are also reported.

3 FMM Accelerated Rank k SVD

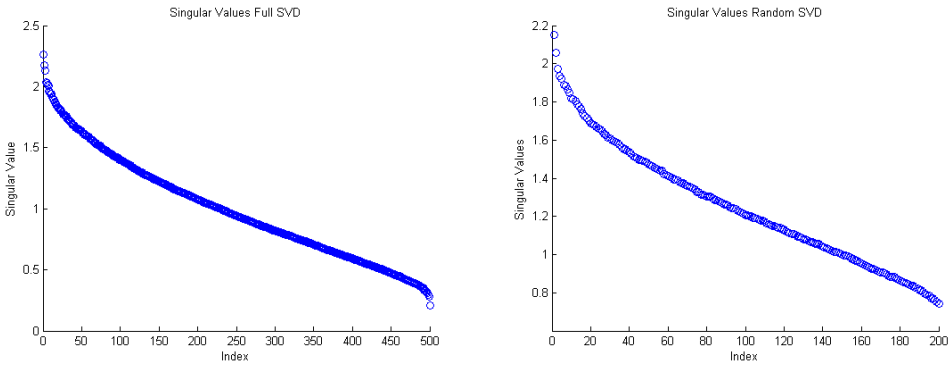
As mentioned previously, given a linear system $Ax = b$, the solution can be found by using the SVD to form a pseudoinverse. The random SVD algorithm can give a rank k approximation of the inverse. And this can be accelerated by using an FMM routine to replace the matrix vector product.

3.1 Implementation

The random SVD algorithm is not natively implemented in MATLAB. Instead of passing in a function handle, it was simple enough to just replace every the matrix vector products involving A with a call to the `mexfunction` created. For any given comparison, the rank and number of matrix-vector products used is kept constant. The source code for the non-FMM and FMM versions of the random SVD can be found in sections 7.1.2 and 7.1.3.

3.2 Results

3.2.1 Singular Values



(a) Full SVD (b) Random SVD: $k = 200, l = 220$

Figure 3.1: Singular Values for $N = 500$

The graphs in Figure 3.1 show the singular values for both the full SVD and the random SVD accelerated via an FMM. The low rank SVD approximation has very similar singular values compared to the ones from the full SVD. This supports the use of our accelerated SVD without a noticeable loss in accuracy.

3.2.2 Error

Type	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Full	6.99e-4	6.20e+0	4.25e-4	3.42e+0	1.41e-1
Random	6.91e-4	5.80e+0	4.21e-4	3.42e+0	8.03e-2
FMM Random	6.91e-4	5.80e+0	4.21e-4	3.42e+0	1.25e+0

Table 3.1: SVD Results: $N = 500$; $k = 200, l = 220$ for Random SVD

Type	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Full	6.97e-4	7.09e+0	4.24e-4	3.64e+0	3.72e+0
Random	6.89e-4	6.67e+0	4.20e-4	3.34e+0	7.48e-1
FMM Random	6.89e-4	6.67e+0	4.20e-4	3.34e+0	4.24e+1

Table 3.2: SVD Results: $N = 2000$; $k = 500, l = 520$ for Random SVD

Type	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Full	7.14e-4	7.70e+0	4.34e-4	3.81e+0	6.68e+1
Random	6.97e-4	6.70e+0	4.25e-4	3.61e+0	5.27e+0
FMM Random	6.97e-4	6.70e+0	4.25e-4	3.61e+0	5.26e+2

Table 3.3: SVD Results: $N = 5000$, $k = 1000, l = 1020$ for Random SVD

Type	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Full	7.41e-4	7.99e+0	4.53e-4	4.21e+0	5.26e+2
Random	7.25e-4	7.54e+0	4.43e-4	4.18e+0	5.52e+1
FMM Random	7.25e-4	7.54e+0	4.43e-4	4.18e+0	5.23e+3

Table 3.4: SVD Results: $N = 10000$, $k = 2500, l = 2520$ for Random SVD

Tables 3.1 to 3.3 also show a similarity in accuracy between the various SVD algorithms. At all values of N, all measures of the error were extremely close. There was practically no difference between the FMM and non FMM routines. And additionally, the Random SVD algorithms were more accurate compared to using the standard MATLAB full SVD. This difference only get larger as N increases.

3.2.3 Time

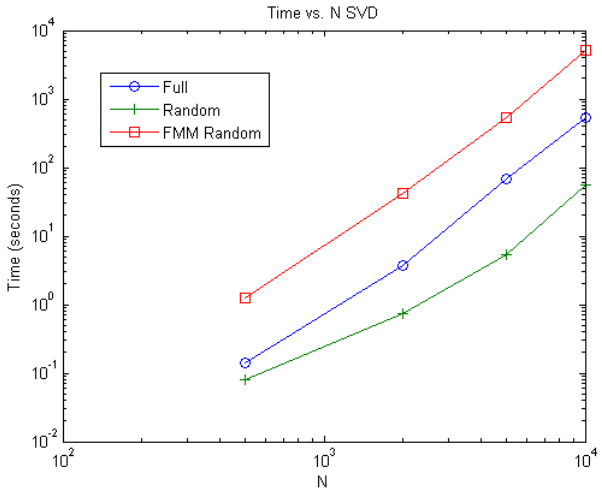


Figure 3.2: Time vs. N

Figure 3.2 shows the time used in solving the system versus N. The routine using the non-FMM random SVD ran faster than the one using a full SVD, but

the FMM version was the slowest of all three. However, looking at the rate of growth of time required, the full SVD routine grew the fasted. At higher values of N, it is expected that the FMM version will eventually run faster.

4 Radial Basis Function Interpolation with the sinc function

4.1 k Parameter

As mentioned previously, the k parameter of the sinc function highly influences the accuracy of the solution. This parameter k determines the width of the interpolating sinc functions. It corresponds to the locality of each function. Higher k's lead to narrower sinc functions, and lower k's lead to wider ones. As k increases, the approximate solution becomes zero everywhere throughout the domain except for the n centers. Obviously A therefore approaches the identity matrix and the system becomes almost already solved. However, this would lead to highly inaccurate approximations through most of the domain. Likewise, if k is too small, the interpolating sinc functions become too global. The accuracy of using the sinc function for RBF interpolation is dependent on finding a good value of k. Preferably, a generally optimal k could be found for a wide range of functions.

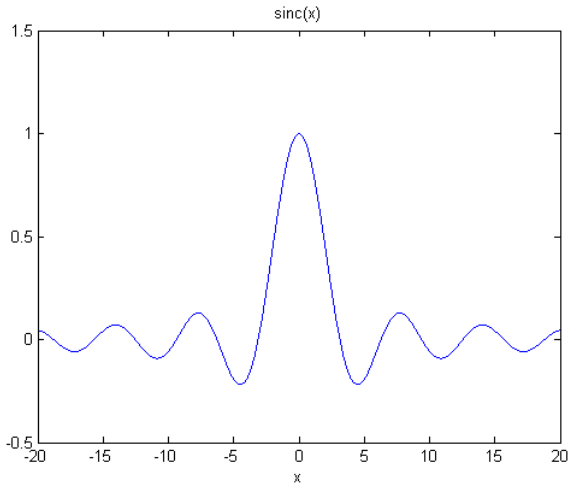


Figure 4.1: Sinc Function

Figure 4.1 shows the sinc function with no k parameter. The zeros occur at every non-zero multiple of π. And the further x is from 0, the smaller the function's amplitude. So given one of the N sampled points, the further an evaluation point is, the smaller the effect it has on the approximation. The points within one period of the evaluation point will have, by far, the largest effect.

It becomes clear that this k parameter will have a significant effect on the accuracy of the resulting interpolant. Figure 4.2 shows an accurate interpolation using a good value of k. However, if k is very large, as in Figure 4.3a, then the interpolating sinc functions are too narrow. The system matrix will be close to the identity matrix and a solution will be quickly found, but the interpolant will be very inaccurate in between sampled points. The opposite happens if k is very small, as in Figure 4.3b. The interpolating sinc functions are too broad and the resulting approximation is not smooth. In addition, the system matrix will have a very poor condition number.

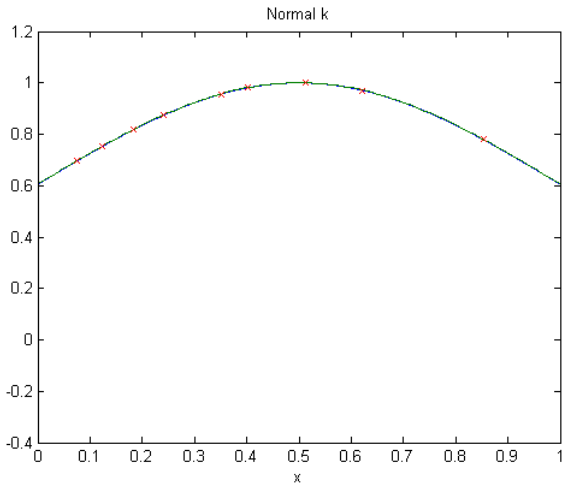
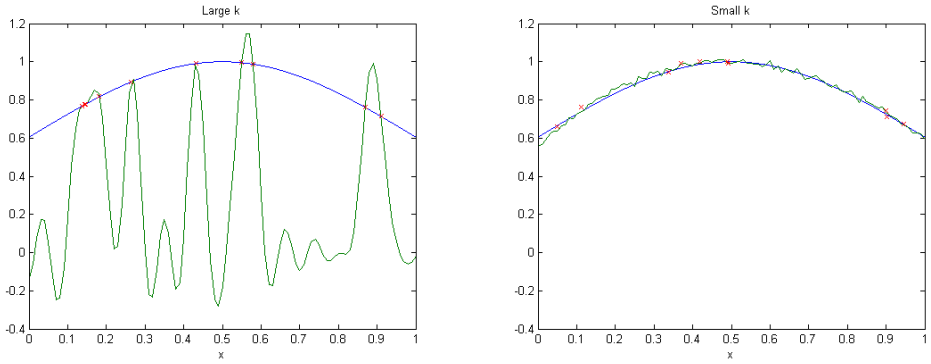


Figure 4.2: Normal k



(a) Large k (b) Small k

Figure 4.3: Large vs. Small k

These tests were done with $N = 10$ data points. Changing N would change how many sinc functions significantly affect any evaluation point. Thus, in testing, k was determined by calculating the average distance to the closest point. In other words, for each of the N points, find the distance to the closest neighboring point and take the average. Then if $k = \frac{2\pi}{a*\text{avgdist}}$, then a will determine how many points lie within the primary wave.

4.2 Results

As mentioned before, too large or small values of the k parameter can lead to very inaccurate results. As Table 4.1 shows, there is a sharp loss of accuracy in the jump from $a = 1$ to $a = 2$ and an even larger one to $a = 4$. There seems to be an exponential increase in error, iterations to convergence, and time needed as a increases.

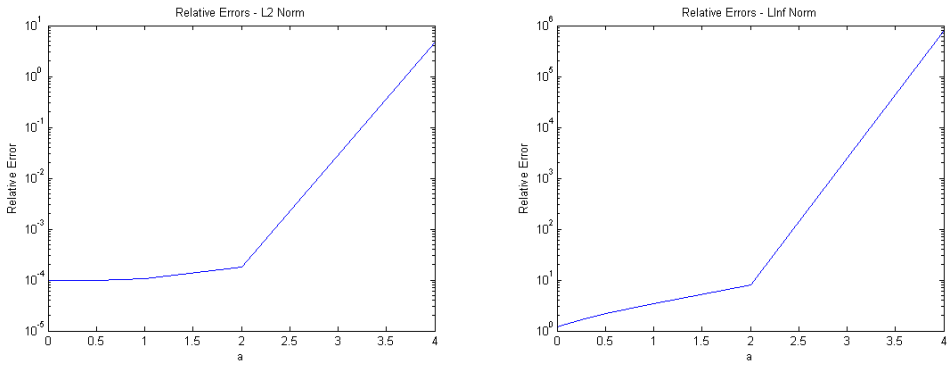
In the methods section, a simple 1D example was given showing that values of k that were too large (equivalent to values of a that are too small) produced even worse results than values of k that were too small. However, the accuracy in these tests seemed to just level off. Table 4.2 shows a similar trend for $N = 1000$.

a	Iters.	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
1/64	3.00	1.62e-4	3.01e+0	9.80e-5	1.21e+0	2.48e-2
1/32	4.00	1.62e-4	3.01e+0	9.80e-5	1.23e+0	9.73e-2
1/16	4.40	1.62e-4	3.19e+0	9.80e-5	1.28e+0	3.33e-2
1/8	5.40	1.62e-4	3.52e+0	9.82e-5	1.39e+0	2.08e-2
1/4	7.40	1.63e-4	3.76e+0	9.85e-5	1.65e+0	3.89e-2
1/2	11.60	1.65e-4	4.59e+0	1.00e-4	2.24e+0	3.96e-2
1	25.40	1.76e-4	7.12e+0	1.07e-4	3.48e+0	8.71e-2
2	109.60	2.94e-4	1.49e+1	1.83e-4	8.03e+0	4.14e-1
4	499.00	7.50e+0	1.34e+6	4.77e+0	8.16e+5	3.92e+0

Table 4.1: Comparing Wavenumber: $N = 500$

a	Iters.	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
1/64	3.20	1.62e-4	3.00e+0	9.80e-5	1.15e+0	1.12e-1
1/32	4.20	1.62e-4	3.04e+0	9.80e-5	1.23e+0	7.29e-2
1/16	4.60	1.62e-4	3.27e+0	9.81e-5	1.28e+0	8.63e-2
1/8	6.40	1.62e-4	3.50e+0	9.82e-5	1.39e+0	8.76e-2
1/4	8.80	1.63e-4	3.78e+0	9.86e-5	1.69e+0	1.12e-1
1/2	14.20	1.66e-4	5.03e+0	1.01e-4	2.35e+0	1.65e-1

Table 4.2: Comparing Wavenumber: $N = 1000$



(a) L_2 Error vs. a (b) L_{inf} Error vs. a

Figure 4.4: Interpolation Error vs. a

5 Iterative Methods and Preconditioning

5.1 Background

Iterative methods are algorithms which produce successively more accurate approximations of the solution. Unlike direct methods, which run until the exact solution is found, iterative methods generally run until the residual reaches a given tolerance. This allows iterative methods to converge to a solution far faster than direct ones can.

For linear systems, there are two main types of iterative methods. Stationary ones are relatively simple but are only guaranteed to converge for certain types of matrices. As such, their use is very limited in practical applications.

5.1.1 Krylov Subspace Methods

The Krylov subspace of order r of a linear system $Ax = b$ is the space spanned by the images of b under the r successive powers of A.

$$\text{span}\{b, Ab, A^2b, \ldots, A^{r-1}b\}$$

For an $N \times nN$ matrix A, its inverse can be represented as a linear combination of its order n Krylov basis.

$$A^{-1} = \frac{(-1)^{N-1}}{\det(A)}\left(A^{N-1} + c_{N-1}A^{N-2} + \cdots + c_1 I_N\right)$$

The iterative methods using this create a Krylov basis with the images of successive residuals. The approximate solution at an iteration step is then found by minimizing the residual over the current subspace. Obviously, this algorithm lends itself to being improved with an FMM routine. Both forming the subspace and minimizing the residual require matrix vector products with A. Improving the efficiency of these will improve the efficiency overall.

For our research, the Generalized minimum residual method (GMRES) was tested. For the nth iteration, this method attempts to minimize the residual in the Kyrlov subspace K_n. This is done by solving a linear least squares problem [9].

5.1.2 Preconditioning

In order to speed up the convergence of the iterative methods described, preconditioners can be used to improve the system's conditioning. Given some preconditioner M, the system $Ax = b$ is transformed into $M^{-1}Ax = M^{-1}b$ for left preconditioning and $AM^{-1}Mx = b$ for left preconditioning.

Generally, a preconditioner M should approximate A so the system $Ax = b$ can approach $x = A^{-1}b$. The closer the preconditioner is to A, the closer the preconditioned system's condition number will be to 1. This would lead to instant convergence. However, if M is chosen to be exactly A, applying the preconditioner is equivalent to solving the original system. Optimal preconditioners must balance improving convergence and ease of calculation. Using the random SVD algorithm allows for a compromise between the two. Since the approximate SVD can be

found for any arbitrary rank k, it can be as expensive or inexpensive as wanted. In addition, as mentioned previously, the lower rank random SVDs are still very accurate approximations.

Our linear system can be simply solved by just using an SVD to estimate the inverse of the matrix A. However, this solution can be improved by combining the random SVD as a preconditioner to an iterative method. This will be more accurate than simply solving the system via the SVD since it will be improved further via reducing the residual. And similarly, it will run faster than simply running the iterative method, since the iterative method will converge much faster.

5.2 Results

5.2.1 GMRES

Type	Iters.	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Standard	23.00	1.77e-4	6.88e+0	1.08e-4	3.48e+0	4.13e-2
FMM	25.40	1.76e-4	7.12e+0	1.07e-4	3.48e+0	8.71e-2

Table 5.1: FMM in GMRES: $N = 500$

Type	Iters.	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Standard	38.80	1.85e-4	8.39e+0	1.13e-4	4.37e+0	2.05e+0
FMM	44.00	1.85e-4	8.59e+0	1.13e-4	4.16e+0	1.97e+0

Table 5.2: FMM in GMRES: $N = 2000$

Type	Iters.	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Standard	48.00	1.90e-4	8.81e+0	1.16e-4	4.70e+0	6.98e+0
FMM	48.00	1.90e-4	8.82e+0	1.16e-4	4.70e+0	5.91e+0

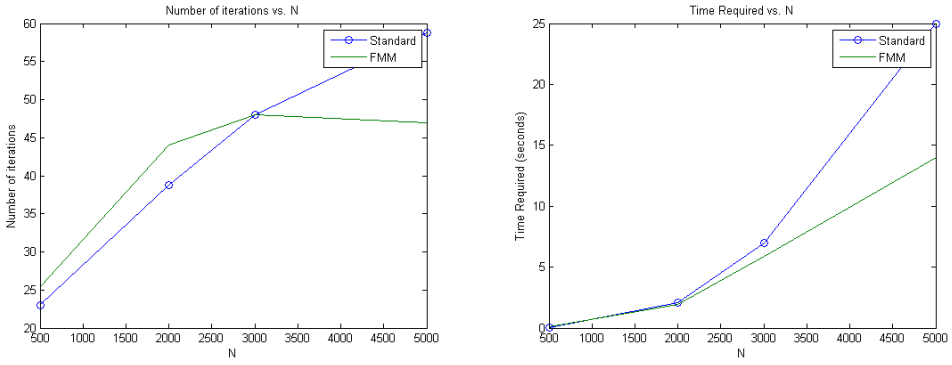
Table 5.3: FMM in GMRES: $N = 3000$

Type	Iters.	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Standard	58.80	1.97e-4	9.87e+0	1.21e-4	5.20e+0	2.50e+1
FMM	47.00	1.98e-4	9.62e+0	1.20e-4	5.20e+0	1.40e+1

Table 5.4: FMM in GMRES: $N = 5000$

At all values of N, GMRES has similar accuracy when using the FMM matrix vector product and when computing the actual product with A. Tables 5.1 through 5.4 show that the errors are very comparable.

However, at smaller values of N, the FMM version takes much longer. At $N = 500$, it requires more than double the time needed by the standard version. This can be attributed to the constant amount of overhead needed by the FMM algorithm, which at smaller values of N, is significant. However, at $N = 2000$, the two versions run in about the same amount of time. And at $N = 3000$, the FMM version runs significantly faster than the standard matrix vector product. As N increases more, this difference is only magnified further.



(a) Number of iterations (b) Time Required

Figure 5.1: Using FMM as a matrix-vector in GMRES

5.2.2 Preconditioning

There were many variations to test for the Random SVD Preconditioner. To serve as a baseline, the standard nonrandom SVD was tested with the same parameters (this is the "Full" type referred to). For the random version of the SVD, there are additional k and l parameters. These refer to the rank of the approximation used and the number of matrix vector products used to calculate this approximation, respectively. For each value of N tested, various combinations of k and l are compared to the full nonrandom SVD and each other.

Type	Iters.	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Full	1.00	1.77e-4	6.88e+0	1.08e-4	3.48e+0	1.32e-2
$k = 500$ $l = 500$	1.00	1.76e-4	6.86e+0	1.07e-4	3.49e+0	1.12e-2
$k = 100$ $l = 100$	5.90	1.91e-4	1.25e+1	1.21e-4	8.97e+0	3.17e-2
$k = 100$ $l = 120$	1.00	1.68e-4	5.08e+0	1.02e-4	2.69e+0	1.05e-2
$k = 200$ $l = 200$	10.75	1.75e-4	6.61e+0	1.06e-4	3.73e+0	5.92e-2
$k = 200$ $l = 220$	1.00	1.70e-4	5.39e+0	1.03e-4	2.84e+0	1.06e-2

Table 5.5: Random SVD: $N = 500$

Looking at the results in Tables 5.5 and 5.6, the preconditioned versions of GMRES were both more accurate and faster than the non preconditioned versions. There are several outliers which have higher errors and which took much longer to run. The randomized SVD algorithm suggests that $k < l < n$ [7]. In addition, $l - k = 20$ was found to be generally optimal. The outliers occurred when $k = l$, so they can be ignored as not indicative of the algorithm's overall performance.

The nonrandom SVD GMRES performed comparably to the non-preconditioned version in accuracy, but consistently converges in 1 iteration. This obviously correlates to a much faster run time too. The random SVD version further improves upon this performance. For $N = 500$, using $k = 100$ and $l = 120$ reduces all

Type	Iters.	L_2 Err.	L_{inf} Err.	L_2 Rel. Err.	L_{inf} Rel. Err.	Time
Full	1.00	1.85e-4	8.35e+0	1.13e-4	4.26e+0	1.75e-1
$k = 2000$ $l = 2000$	1.00	1.86e-4	8.46e+0	1.13e-4	4.30e+0	1.54e-1
$k = 1000$ $l = 1000$	64.00	3.07e-4	3.08e+2	1.88e-4	1.29e+2	4.02e-1
$k = 1000$ $l = 1020$	1.00	1.74e-4	6.35e+0	1.05e-4	3.19e+0	1.55e-1
$k = 500$ $l = 500$	99.80	2.00e-4	1.57e+1	1.23e-4	1.01e+1	5.16e+0
$k = 500$ $l = 520$	1.00	1.70e-4	5.65e+0	1.03e-4	3.11e+0	1.45e-1

Table 5.6: Random SVD: $N = 2000$

error measures, as shown in Table 5.5. Additionally, since a much smaller rank preconditioner is created, it runs much faster. The full interpolation took on average 1.05×10^{-2} seconds, compared to 1.32×10^{-2} for the nonrandom SVD, and 8.71×10^{-2} for the non-preconditioned GMRES.

For larger values of N, the low rank random SVD GMRES takes magnitudes less time than the standard non-preconditioned GMRES.

6 Conclusions

The sinc function showed good results in scattered data interpolation. When evaluated on a tight mesh, the errors were quite low. By tuning the wavenumber, the maximum relative error could be limited to around 1.20. Similarly, the L_2 norm of the errors was also very small. However, the amount of time and number of iterations required for convergence was very high.

Due to the nature of Krylov iterative methods, the main exploding factor is the matrix-vector product. As 5.1b shows, substituting an FMM routine dramatically decreases the growth. At $N = 5000$, the advantages are already dramatic.

The system matrices can also be relatively poorly conditioned. Depending on the wavenumber, 25 iterations or even more could be required. This left a lot of room for improvement in preconditioning. Using the random SVD as a preconditioner reduced the number of iterations required to generally around only 1. While it is clear that high rank approximations would have this effect, lower rank ones did the same. Table 5.5 gives an example of such results. An approximation of rank 100 for $N = 500$ still caused the iterative method to converge in only one iteration in each trial. Since so much fewer iterations are needed, there is a large decrease in the time needed, even at a small $N = 500$. For higher N's, the difference is even larger.

Finally, replacing the matrix-vector products of A in the random SVD routine with an FMM improved the overall performance even further. With a good selection of the approximation rank and number of matrix-vector products used, the time needed could be reduced by more than half. Compared to the standard sinc RBF interpolation, these improvements decreased the time needed by a factor of more than eight.

6.1 Future Directions

While large improvements were made to RBF interpolation with the sinc function, there is still more that could be done. One aspect of GMRES is that every outer iteration uses the same preconditioner. However, a flexible version of GMRES (FGMRES), has been described which allows for a variable preconditioner in each outer iteration. This could allow each iteration to successively better condition the matrix A and converge quicker.

7 Appendices

7.1 Source Code

7.1.1 FMM `mexfunction`

```
1  #include "fintrf.h"
2  C=======================================================================
3  #if 0
4  C
5  C     fantalgo_driver.F
6  C     .F file needs to be preprocessed to generate .for equivalent
7  C
8  #endif
9  C
10 C     fantalgo_driver.f
11 C
12
13 C     This is a MEX-file for MATLAB.
14 C     Copyright 1984-2011 The MathWorks, Inc.
15 C     $Revision: 1.13.2.5 $
16 C=======================================================================
17 C     Gateway routine
18 subroutine mexFunction(nlhs, plhs, nrhs, prhs)
19
20 C     Declarations
21 implicit none
22
23 C     mexFunction arguments:
24 mwPointer plhs(*), prhs(*)
25 integer nlhs, nrhs
26
27 C     Function declarations:
28 mwPointer mxCreateDoubleMatrix, mxGetPr
29 integer mxIsNumeric
30 mwPointer mxGetM, mxGetN
31
32 C     Pointers to input mxArrays:
33 mwPointer points_pr,x_pr,option_pr,param_pr,infoint_pr,inforeal_pr
34
35 C     Pointers to output mxArrays:
36 mwPointer prod_pr
37
38 C     Array information:
39 mwPointer m, n
40 mwSize size
41
42 real*8,allocatable :: points(:,:)
43 real*8,allocatable :: x(:)
44 real*8,allocatable :: prod(:)
45
46 C     Arguments for computational routine:
47 C     Maximum size = numel
48 real*8,allocatable :: sources(:,:), receivers(:,:)
49 integer*8 icall, ierr, option(21), infoint(16)
50 real*8 inforeal(6), param(16)
51 complex*16,allocatable :: monopolestrengths(:), potential(:)
```

```
52
53 C-----------------------------------------------------------------------
54 C     Check for proper number of arguments.
55       if(nrhs .ne. 6) then
56       call mexErrMsgIdAndTxt ('MATLAB:matsq:nInput',
57      +                        'Six inputs required.')
58       elseif(nlhs .ne. 1) then
59       call mexErrMsgIdAndTxt ('MATLAB:matsq:nOutput',
60      +                        'One output required.')
61       endif
62
63 C     Get the size of the input array.
64       m = mxGetM(prhs(1))
65       n = mxGetN(prhs(1))
66       size = m*n
67
68 C     Points array must have 3 rows
69       if(m .ne. 3) then
70       call mexErrMsgIdAndTxt ('MATLAB:matsq:mSize',
71      +                        'Points array must have 3 rows.')
72       endif
73
74 C     Check that the array is numeric (not strings).
75       if(mxIsNumeric(prhs(1)) .eq. 0) then
76       call mexErrMsgIdAndTxt ('MATLAB:matsq:NonNumeric',
77      +                        'Input must be a numeric array.')
78       endif
79
80 C     Create matrix for the return argument.
81       allocate(points(3,n))
82       allocate(x(n))
83       allocate(prod(n))
84       plhs(1) = mxCreateDoubleMatrix(n,1,0)
85       points_pr = mxGetPr(prhs(1))
86       call mxCopyPtrToReal8(points_pr,points,size)
87       x_pr = mxGetPr(prhs(2))
88       call mxCopyPtrToReal8(x_pr,x,n)
89       option_pr = mxGetPr(prhs(3))
90       call mxCopyPtrToReal8(option_pr,option,21)
91       param_pr = mxGetPr(prhs(4))
92       call mxCopyPtrToReal8(param_pr,param,16)
93       infoint_pr = mxGetPr(prhs(5))
94       call mxCopyPtrToReal8(infoint_pr,infoint,16)
95       inforeal_pr = mxGetPr(prhs(6))
96       call mxCopyPtrToReal8(inforeal_pr,inforeal,6)
97       prod_pr = mxGetPr(plhs(1))
98
99 C     Call the computational subroutine.
100      allocate(sources(3,n))
101      allocate(receivers(3,n))
102      sources=points
103      receivers=points
104      allocate(monopolestrengths(n))
105      monopolestrengths(1:n)=x(1:n)
106      allocate(potential(n))
107      icall=0
108      ierr=0
109      call sinc3dfmm_fantalgo_11i(icall,option,param,sources,receivers,
```

```
110 +     monopolestrengths,potential,ierr,infoint,inforeal)
111 icall=2
112 ierr=0
113 call sinc3dfmm_fantalgo_11i(icall,option,param,sources,receivers,
114 +     monopolestrengths,potential,ierr,infoint,inforeal)
115 prod(1:n)=potential(1:n)
116 prod=prod/param(1)
117 icall=3
118 ierr=0
119 call sinc3dfmm_fantalgo_11i(icall,option,param,sources,receivers,
120 +     monopolestrengths,potential,ierr,infoint,inforeal)
121
122 C      Load the data into y_pr, which is the output to MATLAB.
123 call mxCopyReal8ToPtr(prod,prod_pr,n)
124
125 deallocate(points)
126 deallocate(x)
127 deallocate(prod)
128 deallocate(sources)
129 deallocate(receivers)
130 deallocate(monopolestrengths)
131 deallocate(potential)
132
133 return
134 end
```

7.1.2 Random SVD MATLAB Source Code

```
1  function [U,S,V]=RandSVD2(A,k,l)
2      % returns the Randomized SVD of order k for matrix A
3      [m,n]=size(A);
4      G=rand(m,l);
5      R=A'*G;
6      [Ur,Sr,Vr]=svd(R,'econ');
7      Q=Ur(:,1:k);
8      clear Ur Sr Vr;
9      T=A*Q;
10     [U,S,W]=svd(T,'econ');
11     V=Q*W;
12     return
13 end
```

7.1.3 Random SVD using FMM MATLAB Source Code

```
1  function [U,S,V]=RandSVD2FMM(points,option,param,infoint,inforeal,k,l
       )
2  % returns the Randomized SVD of order k for matrix A
3      [m,n]=size(points);
4      G=rand(n,l);
5      R=zeros(n,l);
6      for i=1:l
7      R(:,i)=fantalgo_driver(points,G(:,i),option,param,infoint,
           inforeal);
8      end
9      [Ur,Sr,Vr]=svd(R,'econ');
```

```
10      Q=Ur(:,1:k);
11      clear Ur Sr Vr;
12      T=zeros(n,k);
13      for i=1:k
14      T(:,i)=fantalgo_driver(points,Q(:,i),option,param,infoint,
            inforeal);
15      end
16      [U,S,W]=svd(T,'econ');
17      V=Q*W;
18      return
19  end
```

7.2 Thesis Presentation

Improving Radial Basis Function Interpolation via Random SVD Preconditioners and Fast Multiple Methods

Author: Kerry Cheng
Advisor: Professor Ramani Duraiswami

Fast Multipole Methods (FMM)

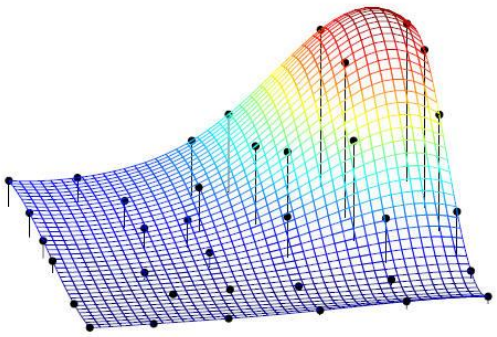
- Mathematical technique developed to speed up calculations of long range forces
- Can speed up sums of type:
  - $s(x_j) = \sum_{i=1}^{N} \alpha_i \phi(\left|\left|x_j - x_i\right|\right|)$
- Splits sum into a close range sum evaluated directly and a long range sum evaluated approximately to a specified accuracy ϵ
- Initially developed for Laplace equation
- Recent research for the Helmholtz equation has given approximation algorithms that compute matrix vector products of specific matrices
- Linear time to any specified accuracy ϵ

Radial Basis Function (RBF) Interpolation

- RBF: Function of one variable, taken as the distance from a center x_i

$$\phi(\left|\left|x_j - x_i\right|\right|)$$

- Approximate multivariable functions, *f,* as linear combination of RBFs at scattered centers (interpolation points)



RBF Interpolation

- Approximation

$$f(x) = \sum_{j=1}^{N} \lambda_j \phi(\left|\left|x - x_j\right|\right|)$$

- Interpolation conditions

$$f(x_i) = \sum_{j=1}^{N} \lambda_j \phi(\left|\left|x_i - x_j\right|\right|)$$

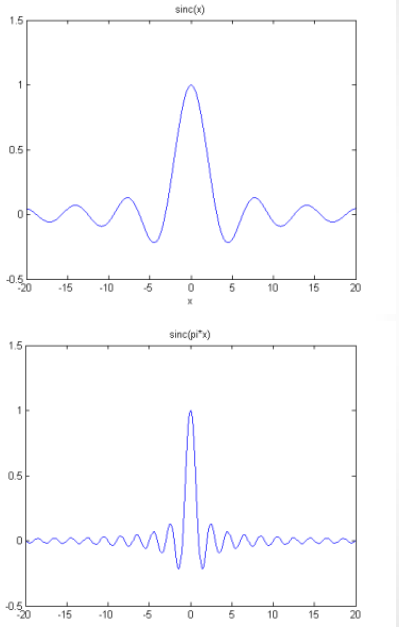
- Leads to linear system to solve $Ax = f$

$$\begin{bmatrix} \phi(\left|\left|x_1 - x_1\right|\right|) & \cdots & \phi(\left|\left|x_1 - x_N\right|\right|) \\ \vdots & \ddots & \vdots \\ \phi(\left|\left|x_N - x_1\right|\right|) & \cdots & \phi(\left|\left|x_N - x_N\right|\right|) \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_N \end{bmatrix} = \begin{bmatrix} f_1 \\ \vdots \\ f_N \end{bmatrix}$$

- Can be solved in variety of ways

Novel Candidate RBF: Sinc Function

- $\phi(x) = \mathrm{sinc}(x) = \frac{\sin(x)}{x}, x \in \mathbb{R}$
- Widely used in signal processing
- Radial basis version:
  - $\mathrm{sinc}(x) = \frac{\sin(k||x-x_*||)}{k||x-x_*||}, x \in \mathbb{R}^d$
  - *k* is a scale parameter
  - Can be expected to affect "influence" of a particular point on interpolant



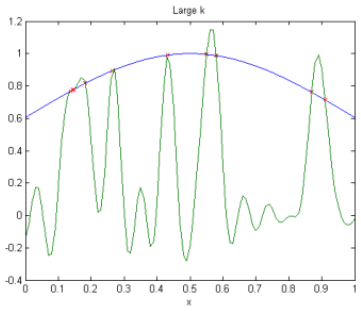
Methods

- N random points chosen in unit cube as a test problem
- Linear RBF system set up and solved for a particular test function
- Approximation evaluated on uniform grid and errors recorded
- Code written in MATLAB with calls to algorithm of sinc FMM code developed by Gumerov and Duraiswami

k Parameter for sinc Function

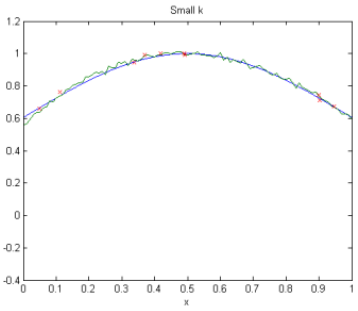
k parameter value is crucial to getting a good interpolation

$N = 10$, Large k



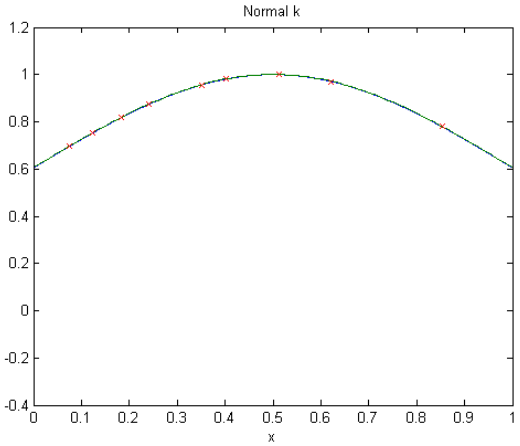
Oscillatory interpolant

$N = 10$, Small k



Noisy interpolant

k Parameter for sinc Function



$N = 10$

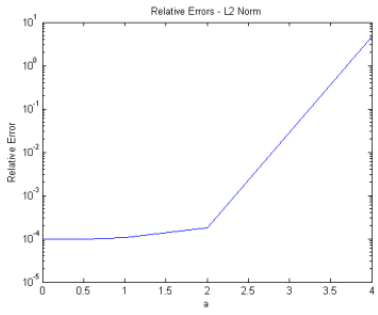
"Good" interpolant

k Parameter heuristic

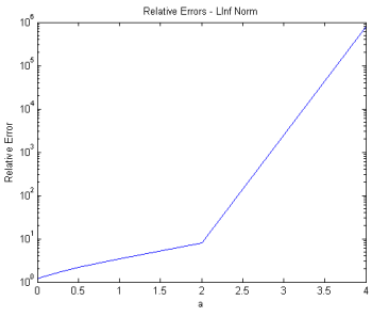
- k determines width of sinc functions
- Developed a heuristic
  - Base k on average distance to closest particle
  - Set $k = \frac{2\pi}{a \cdot \mathrm{avg}dist}$
  - a determines number of points that lie within primary wave of sinc function

k Parameter for sinc Function Results

L_2 Error vs. a



L_{inf} Error vs. a



Solving the System via SVD

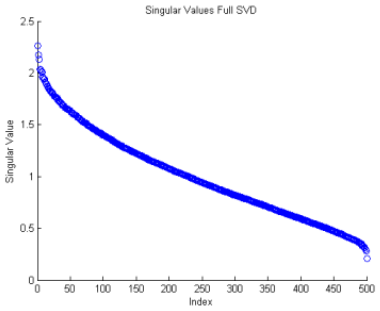
- Singular Value Decomposition (SVD)
- SVD: $A = U\,\Sigma\,V^T$
  - U, V are $N \times N$ matrices with orthonormal columns
  - Σ diagonal matrix with singular values of A on diagonal
- Easy to form pseudoinverse: $A^{-1} = V\,\Sigma^{+}\,U^T$
  - Σ^T is formed by simple replacing every non-zero element with its reciprocal
- Leads to solution:
  - $Ax = F$
  - $A^{-1}Ax = A^{-1}F$
  - $x = A^{-1}F$
- Cost of SVD for a *M*×*N* matrix is O(MN^2) time
- Here *M*=*N*

Randomized SVD

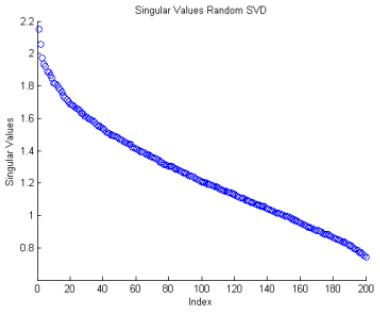
- Standard SVD of an M x N matrix, A, takes $O(MN^2)$ time
- Random SVD algorithm developed by Martinsson et al. constructs rank-k approximation using l random vectors
  - Runs in $O(k^2M + l^2N) + k \cdot C_A + l \cdot C_{A^T}$ time
  - C_A is cost to apply A to arbitrary vector
  - C_{A^T} is cost to apply A^T to arbitrary vector
  - l is k plus a small constant which determines accuracy
- Goal: combine with FMM to give accurate SVD approximation in $O(k \max(M, N))$ time
- Significant improvement in complexity by combining
  - Approximation algorithm (FMM)
  - Randomized algorithm (Random SVD)

Random SVD Results

Full SVD
$N = 500$



Random SVD
$N = 500, k = 200, l = 220$



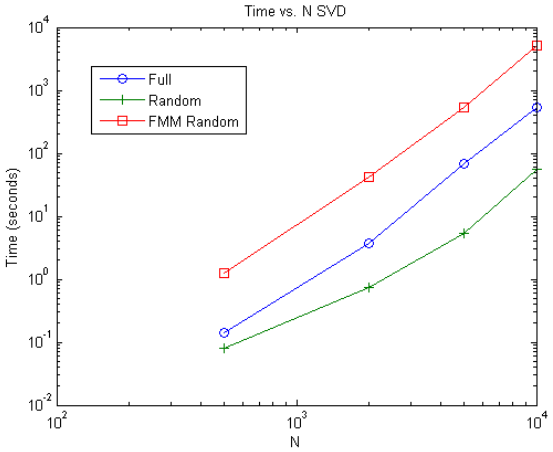
Singular values match well

Random SVD FMM Results

Type	L_2 Error	L_{inf} Error	L_2 Rel. Error	L_{inf} Rel. Error
Full	6.99e-4	6.20e+0	4.25e-4	3.42e+0
Random	6.91e-4	5.80e+0	4.21e-4	3.42e+0
FMM Random	6.91e-4	5.80e+0	4.21e-4	3.42e+0

- $N = 500$
- $k = 200, l = 220$ for Random SVD tests
- Random SVD actually more accurate

Random SVD FMM Results



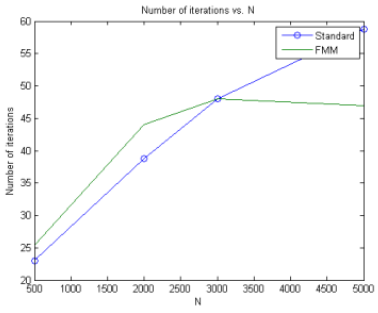
Slope is right (random FMM SVD shows linear scaling), but still expensive
Implementation should be explored in the future

GMRES

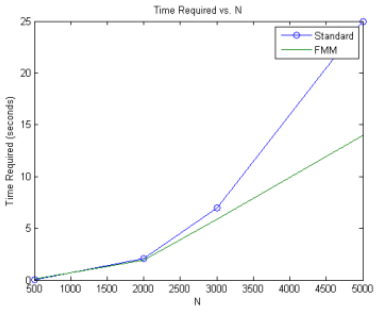
- Iterative method
- Successively produce more accurate approximations of solution
- Krylov subspace of order r for a linear system $Ax = b$
  - $span\{b, Ab, Ab^2, \dots\}$
- At nth iteration, minimizes residual in subspace K_n by solving least squares problem
- Requires many matrix vector products with A

GMRES Results

Number of Iterations to Convergence



Time Required to Solve System



Preconditioning

- Can improve convergence of iterative method by improving conditioning
- Let M be approximation of A
  - $M^{-1}Ax = M^{-1}b$
- Want to use random SVD algorithm as a preconditioner to GMRES iterative method
- More accurate than simply solving via SVD
- Faster than non-preconditioned GMRES

Preconditioning Results Accuracy

Type	L_2 Error	L_{inf} Error	L_2 Rel. Error	L_{inf} Rel. Error
SVD	6.91e-4	5.80e+0	4.21e-4	3.42e+0
No Preconditioning	1.76e-4	6.88e+0	1.07e-4	3.48e+0
Preconditioning	1.70e-4	5.39e+0	1.03e-4	2.84e+0

$N = 500$
$k = 200, l = 220$ for Random SVD
All FMM accelerated

Preconditioning Results Time

Type	Iterations	Time
No Preconditioning	25.4	8.71e-2
Preconditioning	1	1.06e-2

$N = 500$
$k = 200, l = 220$ for Random SVD
All FMM accelerated

Conclusion

- Accurate interpolation
- Choice of sinc wavenumber is broad
- FMM has constant overhead but improves speed of calculating random SVD and GMRES
- Combining random SVD as preconditioner with iterative method gave good results
  - More accurate than directly solving via approximate SVD
  - Faster convergence than non-preconditioned GMRES

Future Directions

- Investigate FGMRES
- Allows variable preconditioner in each iteration
- Each iteration can successively better condition the matrix to converge quicker

Acknowledgements

- Professor Ramani Duraiswami
- Professor Nail Gumerov
- My parents

References

[1] Matlab documentation for gmres function. Website, 2013. `http://www.mathworks.com/help/matlab/ref/gmres.html`.

[2] Matlab documentation for libpointer function. Website, 2013. `http://www.mathworks.com/help/matlab/ref/libpointer.html`.

[3] Brad Baxter. *The Interpolation Theory of Radial Basis Functions*. PhD thesis, University of Cambridge, 2010.

[4] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.

[5] Nail Gumerov and Ramani Duraiswami. Tutorial lecture on the fast multipole method, 2004. Center for Scientific Computing and Mathematical Modeling.

[6] Nail A Gumerov and Ramani Duraiswami. A broadband fast multipole accelerated boundary element method for the three dimensional helmholtz equation. *The Journal of the Acoustical Society of America*, 125:191, 2009.

[7] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis*, 30(1):47–68, 2011.

[8] Estaner Claro Romo and Luiz Felipe Mendes de Moura. Galerkin and least squares methods to solve a 3d convectiondiffusionreaction equation with variable coefficients. *Numerical Heat Transfer, Part A: Applications*, 61(9):669–698, 2012.

[9] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[10] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*. Number 50. Siam, 1997.