Quick Start Guide for mexBBFMM3D, a fast matrix multiplication package for inversion

Bonus: randomized SVD example using mexBBFMM3D

Summary

This package includes instructions and set-up information for using mexBBFMM3D, a method for **fast matrix matrix multiplication**. The package is intended for, but not limited to, use within inversion codes that require multiplication of large covariance matrices with other vectors or matrices. Examples are provided. Also, we include an application of the package to performing matrix decomposition using randomized SVD. Comparison files are provided for showcasing the efficiency of the provided algorithm.

Functions and nomenclature

1. BBFMM3D: Perform fast (linear) multiplication of a kernel matix Q with a vector or matrix: P = QH
2. mexBBFMM3D: Matlab interface for BBFMM3D.
3. randSVD: Performs approximate singular value decomposition of covariance matrix to obtain the first N eigenvectors and eigenvalues
4. randSVD with mexBBFMM3D: Performs approximate singular value decomposition for large covariance kernels Q. BBFMM3D is used for fast multiplication of the Q matrix with random vectors within the randSVD code.

Disclaimer

This is a quick-start guide with instructions on how to set up and use mexBBFMM3D in MATLAB, with two example m-files that can be used to perform matrix-vector and matrix-matrix multiplication for regular and irregular grids, and with examples for running randomized SVD with and without BBFMM3D. A reasonably good knowledge of MATLAB is assumed and minimal understanding of the FMM and randomized SVD theory is needed.

For a full description of the FMM algorithm see [Fong and Darve 2009] in section **Reference**, and for the C++ code, see the BBFMM3D page here. The corresponding code for 2D cases can be found here.

In this guide, we will demonstrate BBFMM3D with an example of multiplication of a Gaussian covariance matrix Q (termed as Gaussian kernel) with a matrix H, and we will then apply randSVD of the Gaussian kernel for a small and large case, with and without BBFMM3D respectively. The methods given here can also be applied for other smooth kernels (see section **Appendix**).

If you use this code, please cite the following paper : Fong, William, and Eric Darve. "The black-box fast multipole methodshod." Journal of Computational Physics 228, no. 23 (2009): 8712-8725.

2. DIRECTORIES AND FILES

```
./example1.m          :   Example of how to use mexBBFMM3D for isotropic,
                          regular grid
./example2.m          :   Example of how to use mexBBFMM3D for anisotropic,
                          irregular grid
./make.m              :   Makefile
./include/            :   Relevant header files
./mexFMM3D.cpp        :   mex function
./eigen/              :   Eigen Library
./BBFMM3D/            :   BBFMM3D library
./README.md           :   This file
./Troubleshooting.md:   Instructions for troubleshooting compilation problems
./RandSVDBBF3D.m      :   Function for performing randSVD for large
                          covariance matrices using BBFMM
./RandSVD.m           :   Function for performing randSVD for small
                          covariance matrices
./cov_reg.m           :   Function to create covariance matrix on a regular grid
./cov_irg.m           :   Function to create covariance matrix for irregular grid
./plotU.m             :   Function to plot U columns obtained from SVD in 3D
```

BBFMM3D

Overview

The Fast Multipole Method (FMM) is an algorithm that performs fast multiplication of an N x N dense matrix Q(x,y) where N is the number of unknown values at points (x,y) in a 2D domain, with a matrix H of size N x m (N>>m). The direct multiplication approach has complexity O(N^2), while the BBFMM3D has linear complexity O(N). The BBFMM3D algorithm uses Chebyshev interpolation to approximate the multiplication. The approximation error can be controlled by the number of Chebyshev nodes used, which can be adjusted to achieve the desired accuracy. mexBBFMM3D provides a Matlab interface for the BBFMM3D package, which is written in C++. Note that the C++ code is faster than the Matlab interface.

The table below shows computation times on a single core CPU when using the mexBBFMM3D package and when using direct multiplication.

N	Time for BBFMM3D (sec)	Time for direct multiplication (sec)
12,500	4.3	22.8
50,000	5.27	367.9
100,000	9.6	1505.3

Setting up mex and compilation

Step 1: Download and be aware of supporting software

You will need the following supporting software:

- The Fast Fourier Transform library, which can be downloaded here. Simply download it for now and read on.
- Matlab SDK files. What you need to do depends on the error messages you will get when following these instructions. See Troubleshooting.md for more.
- Xcode patch for Mac users: If you have upgraded to Xcode 7, see Answer by MathWorks Support Team on 28 Dec 2015 on how to install the patch.

Step 2: Check if you have MEX and MATLAB Symbolic Math Toolbox set up in Matlab

The mexBBFMM3D package relies on MATLAB MEX functions and the MATLAB Symbolic Math Toolbox. In order to use MEX functions, you should setup mex.

Setup MEX by typing the following MATLAB command:

```
mex -setup
```

Once mex is set up successfully, to ensure that MEX is installed, try the following commands:

```
copyfile([matlabroot,'/extern/examples/mex/arraySize.c'],'./','f')
mex -v arraySize.c
arraySize(5000)
```

The last two lines of the screen output should read:

```
Dimensions: 5000x5000
Size of  array in kilobytes: 24414
```

If you do not get this output, take a note of the error you get and refer to file **Trouble Shooting.md** of this package.

Step 3: Download the code from here

A folder called `randSVD-with-BBFMM3D-master` will be downloaded when you click on the above link. Copy the folder `randSVD-with-BBFMM3D-master` to your specific working directory. In MATLAB, set the current folder to be the directory where the code is. You will see the folders `BBFMM3D\` and `Eigen\`, as well as an m-file called (`compilemex.m`) and two example m-files (`example1.m` and `example2.m`) that we will use in this quick start quide. These m-files will be used to compile, set-up, test and use `BBFMM3D` to multiply matrices. You only need to change the input to the example functions. No modifications will be needed to other m-files or the contents of the folder `BBFMM3D` which includes the c++ source code.

Step 4: Compile the MEX file to make sure compilation works

A. Open function `compilemex.m` and read the function description, or type `help compilemex` .

B. Choose input for `compilemex(execname,kerneltype,corlength)` : give your BBFMM3D case a name, e.g. `Test1` and choose your kernel function type, e.g. `GAUSSIAN` and the correlation length. Then compile by giving the command:

```
compilemex('Test1','GAUSSIAN',10)
```

This will compile the source code and generate a MEX-file with the name you provided (e.g. `Test1.mexmaci64`). The extension (`.mexmaci64`) will depend on your platform.

If compilation is successful, you should see the message:

`mex compiling is successful!` .

You can ignore the warnings. If compilation is not successfull, note the error message and refer to file **Trouble Shooting.md** of this package.

Step 5: Run the examples

There are two autonomous examples provided, example1.m for a regular grid, and example2.m for an irregular grid. *The user does not need to change the code.*

If you are only testing the code, we advise that you always operate in the main directory of mexBBFMM3D.

Note for Advanced users:

If you intend to embed the code to your software:

- We advise you to go through the code of the examples and through the documentation of each function and contact us with any questions.
- For the compilation, you have to operate in the main directory of mexBBFMM3D, which contains `make.m`. For use after compilation, you can call the generated MEX-files (e.g., `ExecName.mexmaci64`) by moving them to your own working directory, or add the main directory of mexBBFMM3D to the path.

Example1.m for a regular grid

Function `example1()` uses the mexBBFMM3D code to perform the multiplication of a 3D covariance matrix `Q` defined on a regular grid, with a matrix `H`. The function first compiles the code, and then uses the executable to perform the multiplication.

A. Open file `example1.m` and read instructions, or type `help example1` The example recompiles the code and then computes QH, if on `TestingMode`. If `TestingMode=0` the last compiled executable is used to perform the multiplication. Compilation is always needed when the covariance kernel is changed.

B. Choose input variables for `QH = example1(ExecName,grid,Kernel,corlength,H,TestingMode)`. Avoid using a very large grid while in `TestingMode`, because the code performs the direct multiplication for comparison and it may take a very long time to be completed. Note that for this example the grid must be provided as unique x,y and z locations and (x,y,z) triplets will be created automatically.

Input: - `ExecName` : the name of the mexfile for the Kernel chosen

```
- `grid`     : structure with vectors grid.x, grid.y, grid.z each vector containing x
,y and z coordinates respectively

- `Kernel`   : covariance type, e.g. 'GAUSSIAN'

- `corlength`: correlation length, isotropic anisotropy in z direction supported, see
 code

- `H`        : matrix by which Kernel is multiplied

- `TestingMode`: if set to 1, BBFMM is recompiled and runs in TestingMode in order to
 determine parameters (nCheb) for desired accuracy. if set to 0, the direct multiplic
ation is not performed and the last compiled executable is used for BBFMM3D.
```

Output: - `ExecName.mexmaci64` : executable for mex file for given configuration

```
- `QH`       : Product of Kernel chosen by matrix H specified in input
```

Example usage:

```
 grid.x = -62:4:62; grid.y = -62:4:62; grid.z = -9:3:9;
 QH = example1('TESTNAME',grid,'GAUSSIAN',50,ones(7168,1),1)
```

When run in TestingMode (`TestingMode = 1`), the output will give a relative error that compares the accuracy of BBFMM3D with the direct multiplication of Q*H. Example printout:

```
Starting FMM computation...

Pre-computation time: 1.9583
FMM computing time: 4.6788
FMM total time: 6.6370

Starting direct computation...
Direct calculation starts from: 0 to 0.
Exact computing time: 8.2686
Relative Error: 9.724730e-05
```

If the Relative Error is deemed low enough for your purposes, the code can be run with `TestingMode=0` , in which case the direct multiplication is not performed for comparison. Note that with `TestingMode=0` , the code will use the last compiled executable, even if otherwise specified in the input variables of the example.

Example printout when `TestingMode = 0` :

```
ExecName, Kernel, Corlength ignored
Using executable last compiled
-----------------------------
Executable Name is TESTNAME
Kernel type is GAUSSIAN
Correlation length in x,y,z is 30

Starting FMM computation...

Pre-computation time: 0.0604
FMM computing time: 3.9679
FMM total time: 4.0283
```

Example2.m for an irregular grid with vertical anisotropy

Function `example2()` uses the mexBBFMM3D code to perform the multiplication of a 3D covariance matrix `Q` defined on an irregular grid, with a matrix `H`. The function first compiles the code, and then uses the executable to perform the multiplication. An example irregular grid is provided.

A. Open file `example2.m` and read instructions, or type `help example2` The example recompiles the code and then computes QH, if on `TestingMode`. If `TestingMode=0` the last compiled executable is used to perform the multiplication. Compilation is always needed when the covariance kernel is changed.

B. Choose input variables for `QH = example2(ExecName,grid,Kernel,corlength,H,TestingMode)`. Avoid using a very large grid while in `TestingMode`, as the code performs the direct multiplication for comparison and it may take a very long time to be completed. Q is defined on an **irregular** grid with the option for anisotropy in the z direction. The grid must be provided as (x,y,z) triplets.

The input is as in `example1.m`, with the exception that the grid should now contain all x,y and z triplets, so that each is an `Nx1` vector.

Input:

```
 - `ExecName` : the name of the mexfile for the Kernel chosen

 - `grid`     : structure with vectors grid.x, grid.y, grid.z each vector containing
all x,y and z triplets

 - `Kernel`   : covariance type, e.g. 'GAUSSIAN'

 - `corlength`: correlation length in x and y, isotropic

 - `corlengthz`: correlation length in z

 - `H`        : matrix by which Kernel is multiplied

 - `TestingMode`: if set to 1, BBFMM is recompiled and runs in TestingMode in order t
o determine parameters (nCheb) for desired accuracy.
```

Output:

```
   - `ExecName.mexmaci64`: executable for mex file for given configuration

   - `QH`       : Product of Kernel chosen by matrix H specified in input
```

Example usage:

```
load('./coord_htr.mat')
grid.x = x_htr; grid.y = y_htr; grid.z = z_htr;
QH = example1('TESTNAME',grid,'GAUSSIAN',50,10,ones(23910,1),1)
```

When run in TestingMode (TestingMode = 1), the output will give a relative error that compares the accuracy of BBFMM3D with the direct multiplication of Q*H. The screen printout is the same as in example1 above.

Randomized SVD

Overview

Randomized singular value decomposition is a fast truncated alternative to SVD for large matrices. It is ideally suited for decomposing covariances with fast decaying spectra, or when we are only interested in the first N principal directions of a covariance. The function `RandomizedCondSVD.m` performs the basic randomized SVD, and can be used to decompose any matrix, but is expensive for large matrices (>10000 elements).

Because the randomized SVD algorithm involves the multiplication of the covariance matrix with random

vectors, the BBFMM method can be used to accelerate the decomposition. The function `./RandomizedCondSVDFMM.m` provided here uses BBFMM3D to perform the multiplications required in randSVD. For this reason, it can only be used to perform randSVD for kernels that are comparible with BBFMM3D (see Appendix), but is much faster than the basic implementation of randSVD used in `RandomizedCondSVD.m`.

Basic randSVD

Function structure: `[UN,SN,VN] = RandomizedCondSVD(A,N,q,TestingMode,CompareMode)`

Input:

```
      A:           Covariance matrix. Can be created by cov_reg.m or cov_irg.m
      N:           Number of components of SVD needed, rank of reduced
                   rank svd
      q:           default is 1, or 2 (try 1)
      Testingmode: if 1, the error compared to full svd will be
                   calculated. Caution, do not use for very large matrices as it will
                   take a very long time to perform the full svd
      CompareMode: if 1, the times for different methods will be compared
                   to evaluate efficiency: randomized Svd, full svd, matlab's svd(Q,0
)
                   and matlab's svd(Q,'econ')
```

Example usage:

```
grid.x = -12:6:12; grid.y = -12:6:12; grid.z = -6:3:6;
Q,~]=cov_reg(grid,'GAUSSIAN',6,6,6,[]); imagesc(Q);
[UN,SN,VN] = RandomizedCondSVD(Q,10,1,1);
```

Performance comparison:

m/N	Time for randSVD (sec)	Time for SVD (sec)	Speedup
100/10	0.0018	0.0037	x2.05
200/20	0.0046	0.0105	x2.28
1000/100	0.1192	0.3954	x3.31
2000/200	0.6006	3.4994	x5.82
4000/400	4.5236	28.403	x6.28
8000/800	34.697	249.02	x7.17

randSVD with mexBBFMM3D.

This is the same algorithm as in `RandomizedCondSVD.m`, with all matrix-matrix multiplications performed with mexBBFMM3D and in parallel if there are more than one cores available. The code looks for available processors and splits multiplications in smaller parts to improve efficiency. For a small number of processors the paralellization overhead may reduce efficiency.

This function is presented as an application in which mexBBFMM3D can be used as a black-box algorithm. There are more efficient algorithms to perform fast randomized SVD.

Function structure:
`[U,S,V] = RandomizedCondSVDFMM(grid,Kernel,Corlength,Corlengthz,N,a)`

Input:

```
grid  : structure with vectors grid.x, grid.y, grid.z
        each vector containing all x,y and z coordinates
        respectively
m     : size of covariance matrix number of unknowns
N     : rank of reduced rank svd
a     : oversampling parameter for randSVD
q     : is 1, or 2 (hardcoded below to = 1)
Kernel: covariance type, see compilemex for options
corlength: correlation length in x and y isotropic
corlengthz: correlation length in z
```

Example usage:

```
grid.x = -12:6:12; grid.y = -12:6:12; grid.z = -6:3:6;
gridmesh = CreateRegMesh(grid);
[U,S,V] = RandomizedCondSVDFMM(gridmesh,'GAUSSIAN',100,10,3,9);
```

Performance comparison (In progress):

m/N	Time for randSVD with BBFMM (sec)	Time for basic randSVD (sec)	Speedup
8000/100	146.034	34.697	
10000/100	184.368	(1000) 62.071	
20000/100	1207 ***	(2000) 1001.9**	

** Note: rsvd() developed by Antoine Liutkus (c) Inria 2014 does the same thing in 207 seconds. Investigate. *** Used this before for 24040 with N=30 and less, and utilizing 24 processors so the benefit is more significant.

APPENDIX

Kernel Options

`r` : distance between two points

`L` : length scale parameter

`\sigma^2` : variance parameter

Example of kernel types:

- Gaussian kernel

$$Q(r) = \sigma^2 \exp(-\frac{r^2}{L^2})$$

- Exponential kernel

$$Q(r) = \exp(-\frac{r}{L})$$

- Logrithm kernel

$$Q(r) = A\log(r), A > 0$$

- Linear kernel

$$Q(r) = \theta r, \theta > 0$$

- Power kernel

$$Q(r) = \theta r^s, \theta > 0, 0 < s < 2$$

This package uses:

1. Eigen
2. BBFMM3D

Reference:

1. Sivaram Ambikasaran, Judith Yue Li, Peter K. Kitanidis, Eric Darve, Large-scale stochastic linear inversion using hierarchical matrices, Computational Geosciences, December 2013, Volume 17, Issue 6, pp 913-927 link
2. Judith Yue Li, Sivaram Ambikasaran, Eric F. Darve, Peter K. Kitanidis, A Kalman filter powered by H2-matrices for quasi-continuous data assimilation problems link
3. Saibaba, A., S. Ambikasaran, J. Li, P. Kitanidis, and E. Darve (2012), Application of hierarchical matrices to linear inverse problems in geostatistics, OGST Revue d'IFP Energies Nouvelles, 67(5), 857–875, doi:http://dx.doi.org/10.2516/ogst/2012064. link
4. Fong, W., and E. Darve (2009), The black-box fast multipole method, Journal of Computational Physics, 228(23), 8712–8725.link