

PetFMM

Portable, extensible toolkit for the fast multipole method

User's Manual

Matthew G. Knepley, Felipe A. Cruz, L. A. Barba

September 2009 — Revision 1

Revision History

Name	Date	Notes
Matthew G Knepley, Felipe A Cruz, L A Barba	Sept. 2009	Original Document
xx	2009	Revision xx

Contents

1	Introduction	4
2	Download & Install	4
2.1	Obtaining the code	4
2.2	Building the code	5
3	Test suite	6
3.1	Unit tests	6
3.2	Regression tests	6
4	Algorithm description	6
4.1	Hierarchical space decomposition	8
4.2	Bird's eye view of the complete algorithm	9
5	Using PetFMM in application programs	10
5.1	2D Euler Flow Example	12

1 Introduction

This manual describes PetFMM, a free implementation of the fast multipole method (FMM) in C⁺⁺. It is available to all users, including commercial, and uses the very permissive PETSc license, allowing it to be included in commercial codes. Our aim is to contribute to the work of any and all scientific and engineering research that may benefit from the use of the FMM for accelerated computation of N -body problems.

The first release of PetFMM is only about 3000 lines of C⁺⁺, including comments and blank lines. It can be downloaded from <http://petsc.cs.iit.edu/petsc/>, and we welcome correspondence with potential users or those who wish to extend it for specific purposes.

PetFMM is a portable, extensible fast multipole library for N -body interactions, built on the framework of PETSc. A prominent feature of this parallel library is its capacity for dynamic load balancing, based on an optimizing strategy for distribution of work among processors and a model of both work estimates and communication estimates. Memory estimates are also provided for the user.

2 Download & Install

2.1 Obtaining the code

In order to use PetFMM, you must first download and build PETSc. Installation instructions can be found at

<http://www.mcs.anl.gov/petsc/petsc-as/documentation/installation.html>

Note that you must enable the Sieve package in PETSc, which requires the following configuration options:

```
--with-clanguage=cxx --download-boost --with-sieve
```

In order to run PetFMM in parallel, you will need ParMETIS. PETSc will download and build it for you with the following configuration argument:

```
--download-parmetis
```

or you can point to an existing installation using:

```
--with-parmetis-dir=<path>
```

PetFMM possesses a suite of unit tests. However, in order to run these tests, the user must install the `cppunit` package, <http://sourceforge.net/projects/cppunit>. This package allows flexible configuration of the tests to run, and the method of output generation.

The PetFMM source code can be obtained as a tarball from the repository server at,

<http://petsc.cs.iit.edu/petsc/FMM/archive/tip.tar.gz>

However, we encourage users to consider retrieving the code using the version control program, Mercurial (<http://www.selenic.com/mercurial>). The PetFMM Mercurial repository is located at:

<http://petsc.cs.iit.edu/petsc/FMM>

With Mercurial, users can easily retrieve new patches, upgrade to new releases, and contribute new features or fixes to the codebase. We also make available the development repository at <http://petsc.cs.iit.edu/petsc/FMM-dev> for users who do not wish to wait for new releases in order to try out new features.

Thus, to obtain the source use either

```
wget http://petsc.cs.iit.edu/petsc/FMM/archive/tip.tar.gz .
```

or

```
hg clone http://petsc.cs.iit.edu/petsc/FMM
```

2.2 Building the code

PetFMM can be compiled using

```
cd c++  
make
```

which uses all the compilation settings from PETSC. Specific compilation flags should be introduced into PETSC at the configure stage, as they will be subsequently propagated to PetFMM.

3 Test suite

3.1 Unit tests

Once the user has installed `cppunit`, see Section 2, the unit test can be executed using

```
cd c++
make check
```

The result will be indicated by the output text. If you encounter any errors, please send the full output text to petfmm@barbagroup.bu.edu.

3.2 Regression tests

We also include a set of regression tests to verify the execution on a full, 2D Euler flow problem. The simplest tests, which verify the serial C++ library using the `vortex` driver, can be run using

```
./python/regressionTest.py -p=□
```

Removing the argument also runs a series of parallel tests using the `parallelTest` driver. All test options can be found using the `-help` switch. As before, please send the full output text of any errors to petfmm@barbagroup.bu.edu.

4 Algorithm description

The fast multipole method (FMM) is an algorithm for accelerating computations of the form:

$$f(y_j) = \sum_{i=1}^N c_i \mathbb{K}(y_j, x_i) \quad (1)$$

Such a computation may represent a field value evaluated at point y_j , where the field is generated by the influence of sources located at the set of centers $\{x_i\}$. The sources are often associated with particle-type objects, such as stellar masses, or charged particles. The evaluation of the field at the centers themselves, therefore, represents the well-known N -body problem. In summary: $\{y_j\}$ is a set of evaluation points, $\{x_i\}$ is a set of source points with weights given by c_i , and $\mathbb{K}(y, x)$ is the kernel that governs the interactions between evaluation and source particles. The objective is to obtain the field f at all the evaluation points, which requires in principle $\mathcal{O}(N^2)$ operations if both sets of points have

N elements. Fast algorithms aim at obtaining f approximately with a reduced operation count, ideally $\mathcal{O}(N)$.

The FMM works by approximating the influence of a cluster of particles by a single collective representation, under the assumptions that the influence of particles becomes weaker as the evaluation point is further away, *i.e.*, the kernel $\mathbb{K}(y, x)$ decays as $|x - y|$ increases, and that the approximations are used to evaluate far distance interactions. To accomplish this, the FMM hierarchically decomposes the computational domain and then it represents the influence of sets of particles by a single approximate value. The hierarchical decomposition breaks up the domain at increasing levels of refinement, and for each level it identifies a near and far sub-domain. By using the hierarchical decomposition, the far field can be reconstructed as shown in Figure 1.

Using the computational domain decomposition, the sum in Equation (1) is decomposed as

$$f(y_j) = \sum_{l=1}^{N_{near}} c_l \mathbb{K}(y_j, x_l) + \sum_{k=1}^{N_{far}} c_k \mathbb{K}(y_j, x_k) \quad (2)$$

where the right-most sum of (2), representing the far field, is evaluated approximately and efficiently.

We now need to introduce the following terminology with respect to the mathematical tools used to agglomerate the influence of clusters of particles:

Multipole Expansion (ME): a series expansion truncated after p terms which represents the influence of a cluster of particles, and is valid at distances large with respect to the cluster radius.

Local Expansion (LE): a truncated series expansion, valid only inside a sub-domain, which is used to efficiently evaluate a group of MEs.

In other words, the MEs and LEs are series (*e.g.*, Taylor series) that converge in different sub-domains of space. The center of the series for an ME is the center of the cluster of source particles, and it only converges outside the cluster of particles. In the case of an LE, the series is centered near an evaluation point and converges locally.

As an example, consider a particle interaction problem with decaying kernels, where a cluster of particles far away from an evaluation point is ‘seen’ at the evaluation point as a ‘pseudo-particle’, and thus its influence can be represented by a single expression. For example, the gravitational potential of a galaxy far away can be expressed by a single quantity locally. Thus, by using the ME representing a cluster, the influence of that cluster can be rapidly evaluated at a point located far away —as only the single influence of the ME needs to be evaluated, instead of the multiple influences of all the particles in the

cluster. Moreover, for clusters of particles that are farther from the evaluation point, the pseudo-particle representing that cluster can be larger. This idea, illustrated in Figure 1(b), permits increased efficiency in the computation.

The introduction of an aggregated representation of a cluster of particles, via the multipole expansion, effectively permits a decoupling of the influence of the source particles from the evaluation points. This is a key idea, resulting in the factorization of the computations of MEs that are centered at the same point, so that the kernel can be written as,

$$\mathbb{K}(x_i, y_j) = \sum_{m=0}^p a_m(x_i) f_m(y_j) \quad (3)$$

This factorization allows pre-computation of terms that can be re-used many times, thus increasing the efficiency of the overall computation. Similarly, the local expansion is used to decouple the influence of an ME from the evaluation points. A group of MEs can be factorized into a single LE so that one single evaluation can be used at multiple points locally. By representing MEs as LEs one can efficiently evaluate a group of clusters in a group of evaluation points.

4.1 Hierarchical space decomposition

In order to utilize the tools of MEs and LEs, a spatial decomposition scheme needs to be provided. In other words, for a complete set of particles, we need to find the clusters that will be used in conjunction with the MEs to approximate the far field, and the sub-domains where the LEs are going to be used to efficiently evaluate groups of MEs. This spatial decomposition is accomplished by a hierarchical subdivision of space associated to a tree structure (*quadtree* structure in two dimensions, or an *octree* structure in three dimensions) to represent each subdivision. The nodes of the tree structure are used to define the spatial decomposition, and different scales are obtained by looking at different levels of the tree. A tree representation of the space decomposition allows us to express the decomposition independently of the number of dimensions of the space. Consider Figure 1(a) where a quadtree decomposition of the space is illustrated. The nodes of the tree at any given level cover the whole domain. The relations between nodes of the tree, represent spatial refinement. The domain covered by a parent box is further decomposed into smaller sub-domains by its child nodes. Thus, in the FMM the tree structure is used to hierarchically decompose the space and the hierarchical space decomposition is used to represent the near-field and far-field domains. As an example, consider Figure 1(b) where the near-field for the *black* colored box is represented by the light colored boxes, and the far-field is composed by the dark colored boxes.

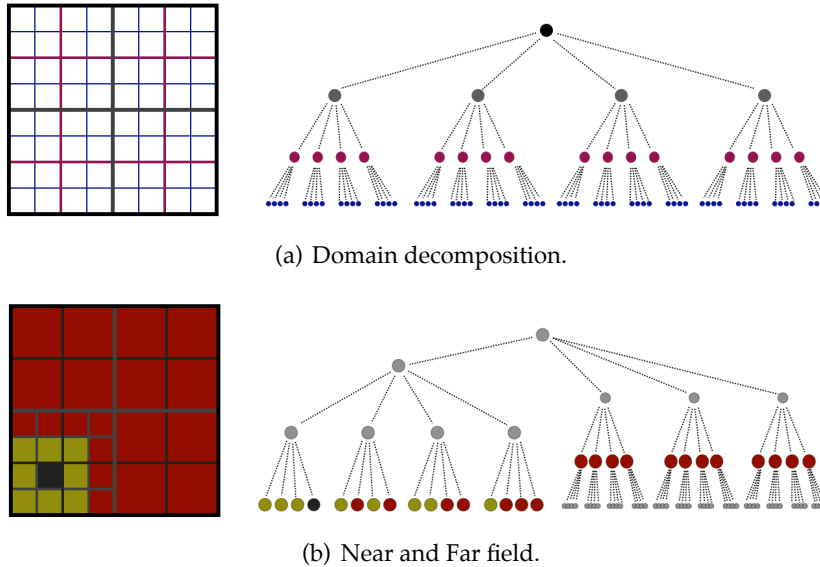


Figure 1: Quadtree decomposition of a two-dimensional domain: (a) presents the hierarchical tree related to the full spatial decomposition of the domain; (b) presents a colored two dimensional spatial decomposition for the black box and its equivalence on the tree. The near-field is composed by the bright boxes and the black box itself, while the far-field is composed by the dark colored boxes. Notice that the far-field is composed of boxes of different levels of the tree structures. The relations between the nodes of the tree simplify the process of composing the near and far domains.

4.2 Bird's eye view of the complete algorithm

After the spatial decomposition stage, the FMM can be roughly summarized in three stages: *upward sweep*, *downward sweep*, and *evaluation step*. In the *upward sweep*, the objective is to build the MEs for each node of the tree. The MEs are built first at the deepest level, level L , and then translated to the center of the parent nodes. Thus, the MEs at the higher levels do not have to be computed from the particles, they are computed from the MEs of the child nodes. In the *downward sweep* of the tree, the MEs are translated into LEs for all the boxes in the *interaction list*. At each level, the interaction list corresponds to the cells of the same level that are in the far field for a given cell. Once the MEs have been translated into LEs, the LEs of upper levels are translated and added up to obtain the complete far domain influence for each box at the leaf level of the tree. At the end of the *downward sweep*, each box will have an LE that represents the complete far-field for the box. Finally, at the *evaluation step*, for every particle at every node at the deepest level of the tree, the final field is evaluated by adding the near-field and far-field contributions. The near field of the particles at a given box is obtained by directly computing the interaction between all the particles in the near domain of the box. The far field of the particles is obtained by evaluating the LE of the box at each particle location.

These ideas can be visualized with an illustration, as shown in Figure 2, which we call the

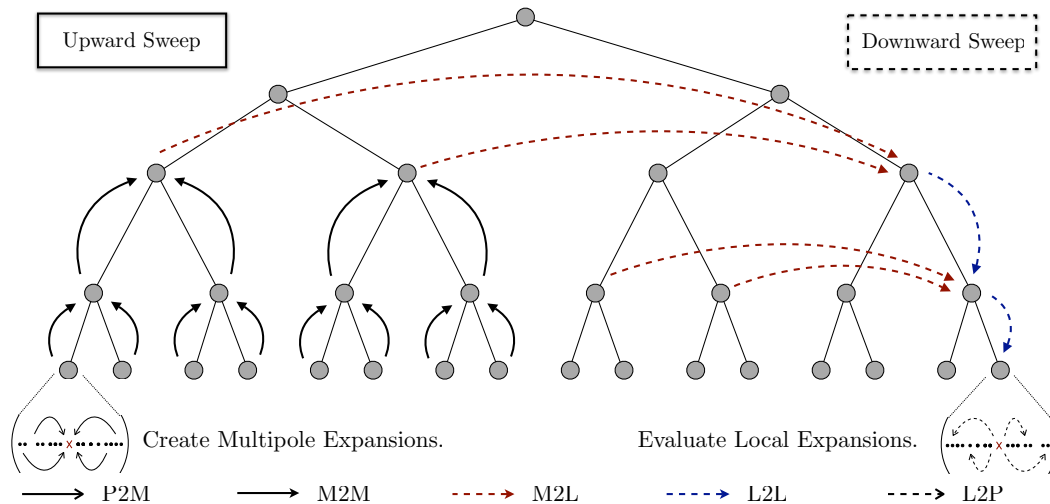


Figure 2: Bird’s eye view of the FMM algorithm. The sketch illustrates the *upward sweep* and the *downward sweep* stages on the tree. Each stage has been further decomposed into the following substages: *P2M*–transformation of particles into MEs (particle-to-multipole); *M2M*–translation of MEs (multipole-to-multipole); *M2L*–transformation of an ME into an LE (multipole-to-local); *L2L*–translation of an LE (local-to-local); *L2P*–evaluation of a LEs at particle locations (local-to-particle).

“bird’s eye view” of the complete algorithm. The importance of this bird’s eye view is that it relates the algorithm computations to the data structure used by the FMM. This will prove to be very useful when we discuss the parallel version that we have developed.

This overview is only intended to introduce the main components of the FMM algorithm, so that we can discuss in the forthcoming sections our strategy for parallelization. Therefore, it is not a detailed presentation (we have left out all the mathematics), and readers interested in understanding all the details should consult the original reference [2], and selections of the abundant literature published since.

5 Using PetFMM in application programs

The primary interaction between an application code and the PetFMM library is through the `Blob` class, which encapsulates all data and methods associated to an individual particle. We currently require a simple, standard interface to which the user may add arbitrary customization. Below is an example of the minimal requirements.

```
typedef evaluator_type::point_type point_type;

class Blob {
public:
```

```

typedef double          circ_type;
typedef std::complex<double> vel_type;
point_type getPosition() const;
public:
  int getNum() const;
  void setNum(const int num);
  void setPosition(const point_type& position);
  circ_type getCirculation() const;
  void setCirculation(const circ_type circulation);
  vel_type getVelocity() const;
  void setVelocity(const vel_type velocity);
  void setVelocity(const double velocity[]);
  void addVelocity(const vel_type velocity);
  void addVelocity(const double velocity[]);
}

```

where `evaluator_type` will be an instantiation of the `FMM::Evaluator<>` template class. The user first defines the `Blob` class appropriate to their problem. They then create a sequence of blobs, meaning an object obeying the STL sequence interface that holds all particles in the problem.

The second task is to define a `Kernel`, or use a predefined kernel in the PetFMM library. The standard `Kernel` interface is as follows:

```

class Kernel {
public:
  typedef Point<dim> point_type;
  typedef Circ_    circ_type;
  typedef Coeff_   coeff_type;
  typedef Vel_     vel_type;
public:
  void precompute();
  void P2MAction(coeff_type C[], const circ_type gamma, const int k,
                const point_type center, const point_type x) const;
  void M2MAction(coeff_type Cdest[], const point_type newCenter,
                const coeff_type Csrc[], const point_type center) const;
  void M2LAction(coeff_type D[], const point_type newCenter,
                const coeff_type interC[], const point_type center) const;
  void L2LAction(coeff_type E[],
                const coeff_type D[], const point_type newCenter,
                const coeff_type parentE[], const point_type center) const;
  void L2PAction(vel_type& u, const int k, const point_type x,
                const coeff_type L[], const point_type center) const;
  template<typename Array, typename BlobIterator, typename NeighborIterator>
  void evaluate(Array& output,
               const BlobIterator& blobBegin, const BlobIterator& blobEnd,
               const NeighborIterator& neighborBlobBegin,
               const NeighborIterator& neighborBlobEnd,

```

```

        const double sigma2, const int k) const;
}

```

Notice that we only require the action of these operators, so that all operations may be carried out in a matrix-free manner, or explicit matrices may be used, depending on the computation. This kernel type, along with the blob type, is used to instantiate the `FMM::Evaluator<>` template class.

In the final step, the user creates a tree of a given number of levels,

```
evaluator_type::tree_type tree(maxLevel+1, debug);
```

and then executes the FMM algorithm, using the blob vorticity values as input and giving velocity values as output

```
evaluator.evaluate(tree, blobs.begin(), blobs.end(), sigma2, k).
```

In serial, the input blobs will contain the output velocity values. However, in parallel, this is not feasible, and the information remains in the Tree object. The user may retrieve it in parallel, from the collection of subtrees, or after calling the `gatherBlobs()` method, may retrieve all the information from the tree on process 0 directly.

5.1 2D Euler Flow Example

In order to simulate the 2D Euler equations using Radial Basis Functions (RBF), a key step is the recovery of a velocity field from a given vorticity field. The vortex driver program solves this problem, following the treatment in [1]. When the vorticity is expressed as a radial basis function expansion, one can always find an analytic integral for the Biot-Savart velocity, resulting in an expression for the velocity at each node which is a sum over all particles. Using the Gaussian basis function, we have:

$$\mathbb{K}_\sigma(x) = \frac{1}{2\pi|x|^2}(-x_2, x_1) \left(1 - \exp\left(-\frac{|x|^2}{2\sigma^2}\right)\right), \quad (4)$$

where $|x|^2 = x_1^2 + x_2^2$. Thus, the formula for the discrete Biot-Savart law in two dimensions gives the velocity as follows,

$$u_\sigma(x, t) = \sum_{j=1}^N \gamma_j \mathbb{K}_\sigma(x - x_j). \quad (5)$$

To begin, we include define a kernel, the Biot-Savart kernel, and a particular evaluator, which in this case is the default serial evaluator:

```
#include <Kernel.hh>
#include <FMM.hh>

class Blob;
typedef FMM::Kernel<double, std::complex<double>, std::complex<double>,
                 NUM_TERMS, DIMENSION, NUM_COEFFICIENTS> kernel_type;
typedef FMM::Evaluator<Blob, kernel_type, DIMENSION> evaluator_type;
```

Here, we have used compile-time constants for the spatial dimension (`DIMENSION`), expansion order (`NUM_TERMS`), and total number of coefficients (`NUM_COEFFICIENTS`). In this problem, the expansion order and number of expansion coefficients is the same, but this will not be true for our 3D electrostatic example.

Next we define the `Blob` class, exactly as shown at the beginning of Section 5. In `main()`, we first create a sequence of blobs distributed inside the domain. For this we use the helper classes `FMM::Utils<>::LatticeSequence`, for regular distributions, and `FMM::Utils<>::SpiralSequence`, for an irregular pattern. The assignment of vorticities to the particles can be constant, using the `ConstantCirculation` class, or follow the analytic solution from Lamb and Oseen, using the `LambOseenCirculation` class. Once this blob sequence is constructed, we have only to call `evaluate()`.

The `evaluate()` function takes a vorticity blob sequence as input, and returns the same sequence with velocity values at the blobs as output. In this serial example, the same sequence can be reused, whereas in parallel we will see that the output is stored in another distributed blob sequence. Thus, we define an evaluator object and its associated tree which partitions the domain,

```
evaluator_type evaluator(PETSC_COMM_SELF, options.debug);
evaluator_type::tree_type tree(options.maxLevel+1, options.debug);
```

and we perform the FMM calculation using the input blobs and basis function parameters

```
evaluator.evaluate(tree, blobs.begin(), blobs.end(),
                  options.sigma*options.sigma, options.k);
```

After the calculation, we can easily check our results by using the direct kernel evaluation to calculate the entire interaction between blobs,

```
std::vector<typename Blob::vel_type> directOutput;
std::vector<Blob> empty;

evaluator.getKernel().evaluate(directOutput, blobs.begin(), blobs.end(),
                              empty.begin(), empty.end(),
                              options.sigma*options.sigma, options.k);
```

where `empty` is just an empty blob sequence. Once we have calculated the exact answer, we can output the true and calculated solution in our verification format

```
std::ostringstream verifyName;  
  
verifyName << options.basename << "fmm." << blobs.size() << ".verify";  
FMM::Output::verify(verifyName.str(), options, tree, blobs, directOutput);
```

which can then also be checked against previous runs (providing a regression test as well).

References

- [1] Felipe A. Cruz, Matthew G. Knepley, and L. A. Barba. Petfmm—a dynamically load-balancing parallel fast multipole library. *submitted to Int. J. Num. Meth. Eng.*, 2009. <http://arxiv.org/abs/0905.2637>.
- [2] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.