The Fast Multipole Method - Implementation Notes

This document gives a quick, hands-on introduction (sec. 1), as well as brief overviews over the structural and design decisions (sec. 2) as well as the process of parallelization (sec. 3) of our implementation of the FMM. A theoretical overview detailing the mathematical and algorithmic background of the method can be found online at `https://github.com/jrotheneder/Mathnotes`.

The programming language used was `C++ 17`, with the only library dependency (besides the `C++ STL` and the `C++ filesystem` library) being the `GNU Scientific Library (GSL)`, which is used for the computation of factorials, binomial coefficients and, in particular, spherical harmonics. For ease of use, an interface to `Wolfram Mathematica 12` also exists, which makes available a subset of the features of the `C++` implementation in a high level language.

1 Examples & Running the Code

Our implementation is header-only. The relevant header files are `balanced_fmm_tree.hpp` for the balanced, and `adaptive_fmm_tree.hpp` for the adaptive algorithm, i.e. either

```
#include "/path/to/lib/balanced_fmm_tree.hpp"
```

or

```
#include "/path/to/lib/adaptive_fmm_tree.hpp"
```

suffice for access to all relevant implementation components, which are part of the namespace `fmm`. The code can be built with the command

```
gcc-8 -Wall -Wno-unknown-pragmas -std=c++17 -pedantic  \
    -O3 -fopenmp helloFMM.cpp -o helloFMM -lstdc++fs -lgsl -lgslcblas
```

We provide a class `Vector_<d>` for d-dimensional vectors, which can be accessed like arrays and constructed with an `std::array<double,d>`, as well as a class `PointSource_<d>` for point sources in d dimensions, constructed by a tuple of a vector and a source strength, as shown below:

```
using namespace fmm;

Vector_<3> v{{1,2,3}};  // double {{}} constructs std::array intermittently
PointSource_<3> ps{v, 4};
```

The FMM algorithm is executed by constructing an instance of `BalancedFmmTree` or `AdaptiveFmmTree`. The constructor has two template arguments, the dimension as well as an optional boolean `field_type`, signalling whether to compute gravitational (if `field_type` is `true`, default) or Coulombic (if `field_type` is `false`) potentials and forces. Its main arguments are

- `sources`, a `std::vector<PointSource_d>`,
- `sources_per_leaf`, `int` specifying the maximal number of sources per leaf,
- `eps`, a double specifying the desired accuracy,
- `force_smoothing_eps` (optional, default 0), a regularization parameter used in N-body simulations.[1]

Once all these parameters are defined, a tree can be built e.g. by

```
AdaptiveFmmTree<d, field_type> fmm_tree(sources, sources_per_leaf, eps,
    force_smoothing_eps);
```

This constructor builds a tree with the desired structure, determines neighbourhood information and executes the up- and downward passes of the FMM. To evaluate potential or force field at a point given as a `Vector_<d>` object `eval_point`, the following calls are required:

```
double potential = fmm_tree.evaluatePotential(eval_point);
Vector_<d> force = fmm_tree.evaluateForcefield(eval_point);
```

The potential energies of and forces on all sources can be acquired in a similar fashion:

```
vector<double> potential_energies = fmm_tree.evaluateParticlePotentialEnergies();
std::vector<Vector_<d>> forces = fmm_tree.evaluateParticleForces();
```

2 Structure & Design

This section gives a rough overview over the ideas and components we consider central to our implementation. It is necessarily brief and we refer the interested reader to the source code and its comments for details. The guiding principle that we attempted to follow in our implementation was maximizing code reuse while providing a simple interface. As discussed in the previous sections, the FMM may be used in multiple configurations: for gravitational and Coulombic interactions, in two and three dimensions and for uniform and non-uniform distributions of sources. The following observations motivated the structure of our implementation:

- The difference between the Coulomb potential and the Newtonian gravitational potential is, in both two and three dimensions, one of sign only, if it is neglected that one is proportional to the charge and the other to

[1] Regularization is used to deal with detrimental effects which the singularity of inverse distance force laws at $\mathbf{x} = \mathbf{x}_0$ has on numerical simulations, which involves [1] "smoothing" the force by introducing a constant factor ε_s as such:

$$\mathbf{E}_{\mathbf{x}_0}(\mathbf{x}) = \begin{cases} \frac{q}{r^2+\varepsilon_s}(\mathbf{x}-\mathbf{x}_0), & \text{in 2D,} \\ \frac{q}{r^3+\varepsilon_s}(\mathbf{x}-\mathbf{x}_0), & \text{in 3D} \end{cases} \tag{1.1}$$

the mass. Implementation wise, it is therefore sensible to represent both point charges and point masses as tuples of a vector and a source strength, compute an electrostatic potential due to these point sources and then, once the electrostatic potential is known wherever desired, switch signs if the gravitational potential is to be returned.

- The difference between the two and three dimensional algorithms is in detail and not structural. By this we mean that, while obvious differences, e.g. in geometry (number & positions boxes in the near neighbour and interactions lists etc.) or in expansion order exist, these differences appear in fixed locations and can be accounted for at compile time. The template mechanism offered by `C++` is ideally suited for this purpose.

- The difference between the balanced and the adaptive algorithm is, however, structural: From tree construction, upward and downward pass and potential evaluation to the information which needs to be stored in nodes and leaves, significant differences between the algorithms exist, strongly suggesting to separate these algorithms.

Based on these observations, we opted for organizing the relevant components into several classes, related by inheritance and shown schematically in fig. 1.

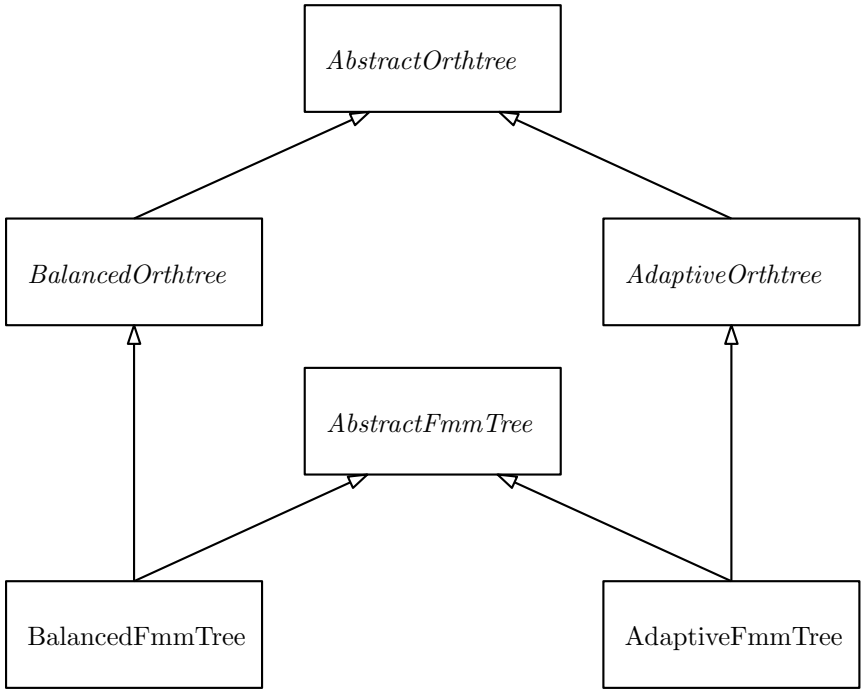

Figure 1: Schematic diagram of the main classes and their inheritance relations. An arrow $B \longrightarrow A$ indicates that class B inherits from ("is-a") class A, italicization denotes abstract classes.

- The class `AbstractOrthtree`[2] defines the tree structure (pointer based) and `BaseNode` class (nested, see fig. 2), captures essential geometric information (height, number of children of a node) allows to query the directions in which the centers of a given box's children lie.

- The classes `BalancedOrthtree` and `AdaptiveOrthtree` provide functions which are required for the construction of balanced and adaptive trees, e.g. for determining which leaf of a balanced tree a given point

[2]The terminology is based on *orthants*, which generalize quadrants and octants. In the literature, the term hyperoctree is used alternatively.

falls into.

- The class `AbstractFmmTree` contains members and methods shared by both the balanced and adaptive versions of the FMM, e.g. computing the extent of the computational domain.

- The classes `BalancedFmmTree` and `AdaptiveFmmTree` are considerable larger than the others and contain Node and Leaf classes (nested) specific to the respective algorithms as well as routines that correspond to the tree building (upward & downward pass) and evaluation algorithms described in the theoretical overview document.

Thee tree structure is, as mentioned, pointer based. Figure 2 shows a class diagram detailing the inheritance relationships between the different types of nodes as well as their most important members for the adaptive case. In particular,

- The `BaseNode` class stores geometric information such as center, extent, parent and children associated with a node and provides methods for checking adjacency of boxes as well as membership (of a point, in the region covered by the node).

- The `FmmNode` class stores neighbour information, in particular also the lists V and X described in the theoretical overview document, as well as the multipole and local expansions associated with every node.

- The `FmmLeaf` class stores additionally the list W as well as the sources contained in a leaf.

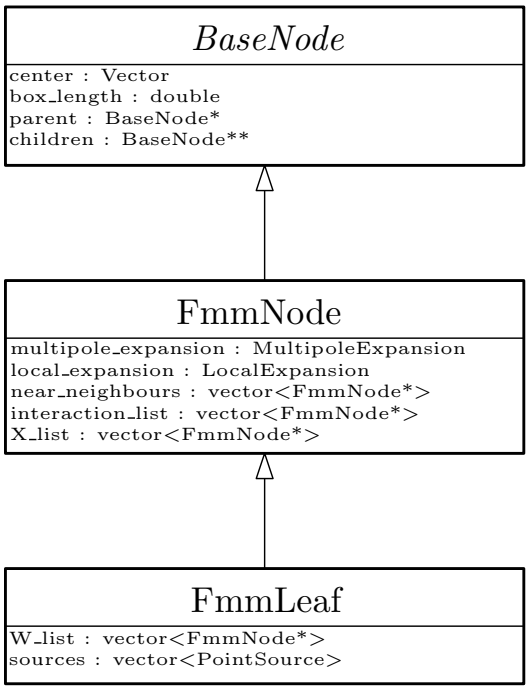

Figure 2: Schematic diagram of the structure of nodes and leaves and their inheritance relations for the adaptive algorithm. An arrow $B \longrightarrow\!\!\triangleright\ A$ indicates that class B inherits from ("is-a") class A, italicization denotes abstract classes and methods. All classes shown in this diagram are nested classes associated with a tree class from fig. 1.

Figure 3 shows a class diagram of the components of the code which realize the multipole and local expansions.

- The abstract class `SeriesExpansion` bundles information and code common to both multipole and local expansions, such as expansion center, order and coefficients, provides a method for adding series expansions w.r.t. the same center and defines an interface for evaluating potential and force due to a series expansion.

- The classes `MultipoleExpansion` and `LocalExpansion` realize the expansions introduced in section the theoretical overview document. They contain the implementations of the various shift and conversion operations, constructors (from sources, from other expansions) and implementations of the methods for evaluation of both potential and force due to the charges represented by the expansion.

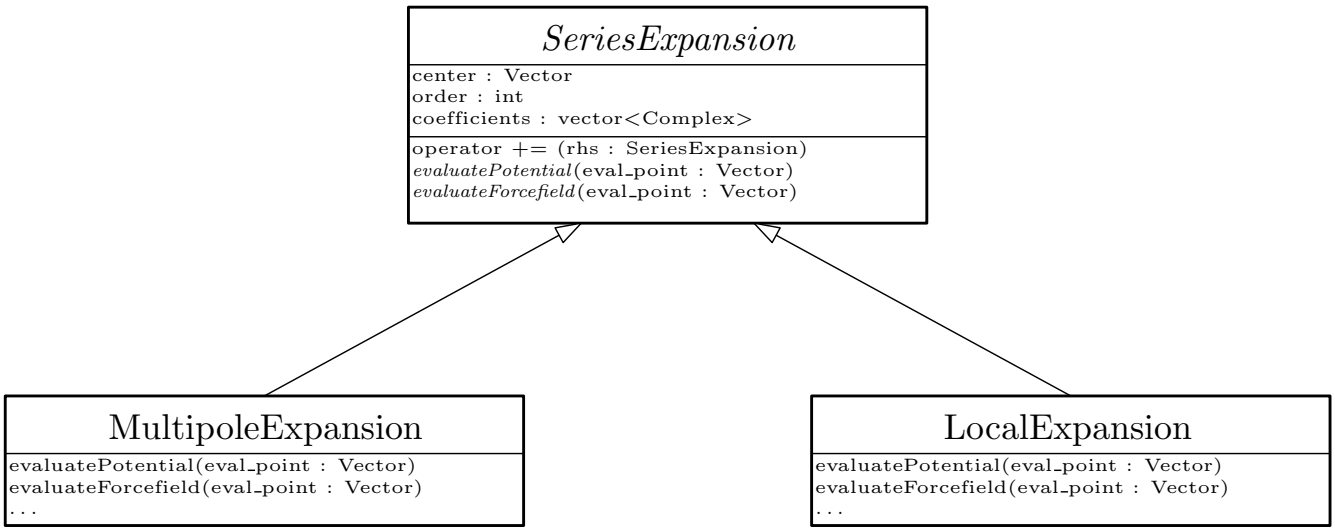

Figure 3: Class diagram of the series expansion interface. An arrow $B \longrightarrow A$ indicates that class B inherits from ("is-a") class A, italicization denotes abstract classes and methods.

3 Parallelism

The sequential implementation was parallelized using `OpenMP` version 4.5. Once the available parallelism is identified, exploiting it is possible in a relatively straightforward manner using `OpenMP`'s loop constructs, provided nodes and leaves are stored in a way which facilitates looping over them level by level.

In the balanced case, this is realized by allocating a contiguous region of memory for the entire tree with the nodes of subsequent levels being stored in consecutive slices of this region, while with the adaptive algorithm, this is realized by splitting nodes level by level and storing nodes level-wise in dynamic length arrays.

References

[1] E Athanassoula et al. "Optimal softening for force calculations in collisionless N-body simulations". In: *Monthly Notices of the Royal Astronomical Society* 314.3 (2000), pp. 475–488.