FMMV

A Fully Vectorized Implementation of the Fast Multipole Method

Harald Hofstätter

AURORA TR200?-??

Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria

E-Mail: `hofi@aurora.anum.tuwien.ac.at`

This work described in this report was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund FWF.

Appendix A

FMMV User's Guide

This user's guide describes the invocation of the FMMV library from user programs written in the programming languages C, Python, or MATLAB. Instructions for building and installing the library can be found in the file `INSTALL` in the distribution directory.

FMMV provides routines based on the Fast Multipole Method (FMM) for the rapid evaluation of the electrostatic (Coulomb) potential and the corresponding force field due to three-dimensional distributions of charged particles and/or dipoles. More precisely, there is a routine `fmmvCoulomb3d` for the approximate evaluation of

$$\phi(\mathbf{x}_i) = \sum_{\substack{j=1 \\ j\neq i}}^{N} \left(\frac{q_j}{\|\mathbf{x}_i - \mathbf{x}_j\|} + \frac{\mathbf{m}_j \cdot (\mathbf{x}_i - \mathbf{x}_j)}{\|\mathbf{x}_i - \mathbf{x}_j\|^3} \right) \quad \text{for } i = 1, \ldots, N. \tag{A.1}$$

Here, q_j and $\mathbf{m}_j$ denote respectively the charge and dipole moment of the particle at $\mathbf{x}_j$. Optionally, `fmmvCoulomb3d` also computes an approximation of the gradient

$$\text{grad}\phi(\mathbf{x}_i), \tag{A.2}$$

which (except for the sign) is the force field at $\mathbf{x}_i$ due to the potential ϕ.

In (A.1) it is assumed that the locations of the charges are also the locations where the potential is to be evaluated. In FMMV the possibilities of optimization due to this additional symmetry are fully exploited, but alternatively it is possible to specify an independent array $\{\mathbf{y}_i\}$ of target locations where the potential shall be evaluated. In this case `fmmvCoulomb3d` computes an approximation to

$$\phi(\mathbf{y}_i) = \sum_{j=1}^{N_{\text{sources}}} \left(\frac{q_j}{\|\mathbf{y}_i - \mathbf{x}_j\|} + \frac{\mathbf{m}_j \cdot (\mathbf{y}_i - \mathbf{x}_j)}{\|\mathbf{y}_i - \mathbf{x}_j\|^3} \right) \quad \text{for } i = 1, \ldots, N_{\text{targets}}. \tag{A.3}$$

A.1 Calling FMMV from C

A.1.1 General Remarks

The C interface of FMMV consists of the routines

```
fmmvCoulomb3d
fmmvCoulomb3d_initialize
fmmvCoulomb3d_evaluate
fmmvCoulomb3d_finalize
fmmvCoulomb3df
fmmvCoulomb3df_initialize
fmmvCoulomb3df_evaluate
fmmvCoulomb3df_finalize
```

where those routines with an additional '`f`' are single precision versions of the corresponding double precision routines without the '`f`'. Programs calling one of these routines should include the header file of FMMV:

```
#include<fmmv_coulomb3d.h>
```

Such programs must be linked to the appropriate FMMV library. On Unix-like systems this is achieved by adding `-lfmmv_coulomb3d -lm` (for double precision) or `-lfmmv_coulomb3df -lm` (for single precision) at the end of the link command.

In the sequel, we only describe the double precision routines. Unless stated otherwise, the corresponding single precision routines behave exactly the same. Of course, for the single precision routines each occurrence of the data type `double` has to be replaced by `float`.

Thread Safety

All routines in FMMV are thread-safe. This means it is safe to call routines in parallel from multiple threads. This holds because except for some precomputed arrays of coefficients which are modified only at the first invocation of the library, no global data is shared across internal subroutines of the library. In particular, the routines `fmmvCoulomb3df_initialize`, `fmmvCoulomb3df_evaluate`, and `fmmvCoulomb3df_finalize`, which by necessity have to share some data amongst them, do so by sharing only a handle, which has to be passed as an argument, see Section A.1.3. However, sharing the same handle across multiple threads is not allowed.

Error Handling

Generally, the routines of FMMV are functions with a return value of type `char*` (pointer to char). On success, they return a null pointer. On failure, they return a pointer to a string, containing a meaningful error message. It is strongly recommended, always to check this return value and in case of failure to display the error message and abort the program. Note that this is *not* done by default.

A.1.2 `fmmvCoulomb3d`

The FMMV library is most straightforwardly invoked from C code by calling the routine `fmmvCoulomb3d` (`fmmvCoulomb3df` for single precision). This is the recommended way if for a given distribution of particles, the potential shall be evaluated only once for a fixed set of charges or dipole moments. For the case of several sets of charges, see Section A.1.3.

Function Declaration

```
char* fmmvCoulomb3d(
    unsigned int NParticles,
    double particles[][3],
    double charges[],
    double dipoleMoments[][3],
    unsigned int NTargets,
    double targets[][3],
    double pot[],
    double grad[][3],
    struct FmmvOptions *options,
    struct FmmvStatistics *statistics);
```

Arguments

- `NParticles` is the number of particles or the number of sources if independent targets are specified.

- `particles` is a pointer to an array containing the coordinates of the particles. This means that the pointer `particles` points to a memory block large enough to contain at least 3*`NParticles` elements of type `double`. The arrangement of the elements is assumed to be the obvious one:

0: x-coordinate of particle #0
1: y-coordinate of particle #0
2: z-coordinate of particle #0
3: x-coordinate of particle #1
⋮
3*NParticles-1: z-coordinate of particle #NParticles−1

The array `particles` can be allocated either on the stack using

```
double particles[NParticles][3];
```

or on the heap using

```
double (*particles)[3];
...
particles = ((double(*))[3]) malloc(NParticles*3*sizeof(double));
```

The particular form of these declarations ensures that the j-th Cartesian coordinate of the i-th particle ($j \in \{0, 1, 2\}$, $i \in \{0, 1, \ldots,$ `NParticles`$-1\}$) can conveniently be accessed to by `particles[`i`][`j`]`.[1]

Analogous considerations apply to the arrays `dipoleMoments`, `targets`, and `grad`.

- `charges` is a pointer to an array such that `charges[`i`]` is the charge of the i-th particle.
- `dipoleMoments` is a pointer to an array such that `dipoleMoments[`i`][`j`]` is the j-th Cartesian component of the dipole moment of the i-th particle.

  At most one of the pointers `charges` and `dipoleMoments` may be a null pointer:

  `dipoleMoments==0`: all particles are monopole sources, i.e. $\mathbf{m}_j = 0$ for all j in (A.1) or (A.3);

  `charges==0`: all particles are dipole sources, i.e. $q_j = 0$ for all j in (A.1) or (A.3).
- `NTargets` is the number of target locations. `NTargets` is ignored if `targets` is a null pointer.

[1] Note that in Fortran the order would be the opposite way round: To get the same arrangement of elements, the array would have to be defined by

```
DOUBLE PRECISION PARTICLES(3, NPARTICLES)
```

The j-th coordinate of the i-th particle is accessed by `PARTICLES(`j`,`i`)`, where $j \in \{1, 2, 3\}$ and $i \in \{1, 2, \ldots,$ `NPARTICLES`$\}$.

- `targets` is a pointer to an array such that `targets[`i`][`j`]` is the j-th Cartesian coordinate of the i-th target location, i.e. the location where the potential (and optionally the gradient) shall be evaluated.

  If `targets` is a null pointer, then `particles` is used for the target locations, and an approximation to (A.1) instead of (A.3) is computed.

- `pot` is a pointer to an array of size at least `NTargets` (or `NParticles` if `targets` is a null pointer) elements of type `double`. Note that `pot` must not be a null pointer. After successful execution of `fmmvCoulomb3d`, `pot[`i`]` contains the potential at the i-th target location.

- `grad` is a pointer to an array of size at least 3*`NTargets` (or 3*`NParticles` if `targets` is a null pointer) elements of type `double`. If `grad` is a null pointer no gradients are computed. Otherwise, after successful execution of `fmmvCoulomb3d`, `grad[`i`][`j`]` contains the j-th Cartesian component of the gradient at the i-th target location.

- `options` is a pointer to a `FmmvOptions` structure for the fine-tuning of the algorithm, see Section A.1.4 below. If `options` is a null pointer default options are used.

- `statistics` is a pointer to a `FmmvStatistics` structure, cf. Section A.1.5 below. If `statistics` is a null pointer no statistics are provided. Otherwise, `statistics` contains e.g. timing and memory usage information collected during the execution of the code.

Example of Usage

```
#include<stdio.h>    /* stderr, fprintf */
#include<stdlib.h>   /* exit */
#include<fmmv_coulomb3d.h>
...
#define N_PARTICLES 10000
...
{
    char *error_message;
    double particles[N_PARTICLES][3];
    double charges[N_PARTICLES];
    double pot[N_PARTICLES];
    ...
    /* some code for filling the arrays particles and charges */
    ...
    error_message = fmmvCoulomb3d(
        N_PARTICLES, particles, charges,
```

```
            0,      /* no dipoles */
            0, 0,   /* no independent target locations */
            pot,
            0,      /* don't compute gradient */
            0,      /* use default options */
            0);     /* don't generate statistics */

        if (error_message) {
            fprintf(stderr, "%s\n", error_message);
            exit(1);
        }
        ...
    }
```

A.1.3 `fmmvCoulomb3d_initialize`, `fmmvCoulomb3d_evaluate`, `fmmvCoulomb3d`

Equation (A.1) (with $\mathbf{m}_j = 0$ for simplicity) can be written as

$$y = Ax, \tag{A.4}$$

with the vector $x \in \mathbb{R}^N$ of charges q_j, the (symmetric) matrix $A \in \mathbb{R}^{N \times N}$ given by

$$A_{ij} = \frac{1}{\|\mathbf{x}_i - \mathbf{x}_j\|}, \tag{A.5}$$

and the result vector $y \in \mathbb{R}^N$ of the potentials $\phi(\mathbf{x}_i)$. In many applications, however, the vector y is given and the equation (A.4) has to be solved for the vector x. This is usually done by some iterative method, in course of which many matrix-vector products of the form (A.4) have to be computed.

To handle such cases efficiently, FMMV provides separate routines for the three main steps—initialization, evaluation, and finalization—of the algorithm:

- `fmmvCoulomb3d_initialize` performs in advance all initializations (like building up data structures, precomputing coefficients, etc.) depending only on the matrix A, i.e., on the geometric distribution of the particles $\mathbf{x}_i$.

- Subsequent calls to `fmmvCoulomb3d_evaluate` evaluate (A.4) for several vectors x without performing these initializations again.

- Finally, `fmmvCoulomb3d_finalize` frees all resources that have been allocated by `fmmvCoulomb3d_initialize`.

Function Declarations

```
char* fmmvCoulomb3d_initialize(
    void** fmmvHandle,
    unsigned int NParticles,
    double particles[][3],
    unsigned int NTargets,
    double targets[][3],
    int typeSources,
    int computeGradient,
    struct FmmvOptions *options,
    struct FmmvStatistics *statistics);

char* fmmvCoulomb3d_evaluate(
    void* fmmvHandle,
    double charges[],
    double dipoleMoments[][3],
    double pot[],
    double grad[][3],
    struct FmmvStatistics *statistics);

char* fmmvCoulomb3d_finalize(
    void* fmmvHandle,
    struct FmmvStatistics *statistics);
```

Arguments

- `fmmvHandle` serves as a handle for the internal communication between the routines. It is declared as a generic pointer (`void*`) pointing to some internal data structure hidden to the user. Note that `fmmvHandle` is an output parameter for `fmmvCoulomb3d_initialize` (causing the data type `void**` in its declaration), whereas it is an input parameter for the other routines.

- `NParticles`, `particles`, `NTargets`, `targets`: These arguments specifying the distribution of the source and target locations have the same meaning as for `fmmvCoulomb3d`, cf. Section A.1.2. Note that these data have to be supplied only for `fmmvCoulomb3d_initialize`.[2] Again, `target` may be a null pointer, in which case the points in `particles` are used for the target locations.

[2]The arrays `particles` and `targets` are sorted by `fmmvCoulomb3d_initialize`, and `fmmvCoulomb3d_evaluate` can access to copies of these sorted arrays through `fmmvHandle`.

- `typeSources` denotes the kind of sources to be handled:

  `typeSources==0`: only monopole charges;

  `typeSources==1`: only dipoles;

  `typeSources==2`: monopoles and dipoles.

  Note that according to `typeSource` the arguments `charges` and `dipoleMoments` of `fmmvCoulomb3d_evaluate` may become mandatory, i.e., they must be pointers to appropriate arrays and may not be null pointers.

- `computeGradient` indicates whether gradients shall be computed. If the quantity `computeGradient` is non-zero, the argument `grad` of `fmmvCoulomb3d_evaluate` is mandatory and must be a pointer to an appropriate output array.

- `charges` and `dipoleMoments` have the same meanings as in the case of `fmmvCoulomb3d`, cf. Section A.1.2. According to the value of the parameter `typeSources` of `fmmvCoulomb3d_initialize`, each of these arguments of `fmmvCoulomb3d_evaluate` is either mandatory or ignored.

- `pot` and `grad`: Cf. Section A.1.2. If the parameter `computeGradient` of `fmmvCoulomb3d_initialize` was zero, `grad` is not used and may be a null pointer.

- `options` and `statistics`: Cf. Section A.1.2.

Example of Usage

```
#include<stdio.h>    /* stderr, fprintf */
#include<stdlib.h>   /* exit */
#include<fmmv_coulomb3d.h>
...
#define N_PARTICLES 10000
...
{
    void *fmmvHandle;
    char *error_message;
    double particles[N_PARTICLES][3];
    double charges[N_PARTICLES];
    double pot[N_PARTICLES];
    ...
    /* some code for filling the array particles */
    ...
    error_message = fmmvCoulomb3d_initialize(
        &fmmvHandle, N_PARTICLES, particles,
```

```
        0, 0,  /* no independent target locations */
        0,     /* only monopoles */
        0,     /* don't compute gradient */
        0,     /* use default options */
        0);    /* don't generate statistics */

    if (error_message) {
        fprintf(stderr, "%s\n", error_message);
        exit(1);
    }
    ...
    /* some code for filling the array charges */
    ...
    error_message = fmmvCoulomb3d_evaluate(
        fmmvHandle, charges, 0, pot, 0, 0);

    if (error_message) {
        fprintf(stderr, "%s\n", error_message);
        exit(1);
    }

    error_message = fmmvCoulomb3d_finalize(
        fmmvHandle, 0);

    if (error_message) {
        fprintf(stderr, "%s\n", error_message);
        exit(1);
    }
    ...
}
```

A.1.4 Fine-Tuning of FMMV

For the fine-tuning of the algorithm a pointer to an options structure with the following declaration may be supplied to the routines `fmmvCoulomb3d` and `fmmvCoulomb_initialize`:

```
struct FmmvOptions {
    int precision;           /* default 0 */
    double scale;            /* default 1.0 */
    int splitThreshold;       /* default depends on context */
    int splitTargetThreshold; /* default depends on context */
```

```
    int maxLevel;            /* default depends on context */
    int maxTargetLevel;      /* default depends on context */
    int directEvalThreshold;  /* default depends on context */
    int periodicBoundaryConditions;  /* default 0 */
    int useHilbertOrder;     /* default 0 */
    int useApproxInvSqrt;    /* default 0 */
    int useFarfieldNearfieldThreads; /* default 0 */
    int PAPIeventSet;        /* default PAPI_NONE */
};
```

A negative value for a field of this structure means that the default value indicated in the corresponding comment shall be taken. For convenience, FMMV provides the routine

```
struct FmmvOptions fmmvGetDefaultOptions(void);
```

which returns an options structure with all fields set to -1. This makes it easy to specify only a subset of all possible options. For example, in the following code segment only the options `precision` and `periodicBoundaryConditions` are explicitly specified, all other options are set to their default values:

```
...
struct FmmvOptions options;
...
options = fmmvGetDefaultOptions();
options.precision = 1;
options.periodicBoundaryConditions = 1;
...
```

The available options, sorted by categories, are listed in the following.

Options for Controlling Precision

- `precision`

  The accuracy of the fast multipole method implemented in FMMV is determined by the parameters p (length of multipole expansions) and s (order of a certain internally used quadrature formula, for details see Chapter **??**). The value of `precision` selects one of the three combinations of p and s, which are currently implemented in FMMV, according to the following table:

`precision`	p	s	ε
0	6	8	10^{-3}
1	16	17	10^{-6}
2	23	26	10^{-9}

Here ε is approximately the expected accuracy. `precision==2` is not implemented for single precision, because when calculations are carried out in single precision, no accuracy higher than $\varepsilon = 10^{-6}$ can be expected.

- `useApproxInvSqrt`

  The value of `precision` affects only the accuracy of the calculation of far field interactions. Interactions in the near field are directly calculated by (A.1) or (A.3) using an implementation of the function $x \mapsto 1/\sqrt{x}$, which calculates its result to full floating-point precision. Some processors however, like, e.g., Pentium processors with SSE extensions or PowerPC processors with AltiVec extensions, provide approximations to $x \mapsto 1/\sqrt{x}$ in hardware, which—if available—can be utilized in FMMV by setting `useApproxInvSqrt` to a value ≥ 1. Such approximations are currently available only in single precision. Their utilization leads to a significant decrease of the run time, but their accuracy is compatible with `precision==0` only, i.e., $\varepsilon = 10^{-3}$.

Options for Controlling the Oct-Tree Data Structure

- `splitThreshold`
- `maxLevel`
- `splitTargetThreshold`
- `maxTargetLevel`

  `splitThreshold==0` requires the oct-tree data structure to be built according to the *non-adaptive FMM*. Boxes are recursively divided into eight child boxes until `maxLevel` levels of boxes have been obtained. Leaf boxes occur only at the highest (finest) level. For the non-adaptive FMM the values of `splitTargetThreshold` and `maxTargetLevel` are ignored, even if independent target locations are specified.

  `splitThreshold>0` requires the oct-tree data structure to be built according to the *adaptive FMM*. Boxes that are not already in level `maxLevel` are recursively divided into eight child boxes as long as they contain more than `splitThreshold` particles. Leaf boxes may occur at all levels. If independent target locations are specified, the options `splitThreshold` and `maxLevel` only apply to the assignment of sources to source boxes. Assignment of targets to target boxes is controlled analogously by the options `splitTargetThreshold` and `maxTargetLevel`.

  It should be mentioned that for `splitThreshold==0` and `maxLevel==0` the FMM is not involved at all, all interactions are computed directly. This may

be important for comparison purposes, because for this direct algorithm the same highly optimized routines of FMMV are used. Usually, not much effort is spent on the optimization of the direct algorithm, which makes the comparison with the FMM somewhat unfair.

- `useHilbertOrder`

  While building the oct-tree data structure, all particles are sorted in such a way that for each box the particles belonging to this box are contiguously stored in memory. By default, this goal is achieved by sorting the particles according to standard *Molton order*, cf. Chapter **??**.

  Alternatively, setting `useHilbertOrder` to a value ≥ 1 requires the particles to be sorted along a *Hilbert space filling curve*, which often leads to better performance due to improved cache utilization.

- `scale`

  Before building the oct-tree data structure, all particles are scaled to fit into $[0,1]^3$, i.e., the unit cube.[3] An additional scaling factor $\in (1/2, 1]$ may be specified via `scale`. For the non-adaptive FMM this has the effect, that for a given number of levels, less leaf boxes are generated and accordingly more particles belong to each leaf box. Bearing in mind that for uniform particle distributions the average number of particles per leaf box is an essential parameter to be optimized, this additional scaling allows much finer control of this parameter than by controlling it only by means of the number of levels. More information on this idea can be found in [2].

Far Field/Near Field Parallelism

- `useFarfieldNearfieldThreads`

  By the very nature of the FMM, the two steps

  - calculating all far field interactions and
  - calculating all near field interactions

  are essentially mutually independent. Thus, the order in which they are performed does not matter, they can even be performed in parallel. As experience shows, balancing computing time among these two steps often minimizes the total computing time. Consequently, doing these steps in parallel seems even more desirable. In FMMV this parallelism is implemented using POSIX threads, and is enabled by setting `useFarfieldNearfieldThreads` to a value ≥ 1. Of course, this only makes sense if more than one processor (or a dual core processor) is available.

[3] Actually the particles are scaled to fit into the cube $[1/2, 1]^3$, which offers some technical advantages, see Chapter **??**.

Periodic Boundary Conditions

- `periodicBoundaryConditions`

  FMMV supports the use of periodic boundary conditions. If `periodicBoundaryConditions` is set to a value ≥ 1, the appropriate routines of FMMV compute an approximation to

$$\phi(\mathbf{x}_i) = \sum_{\mathbf{n}\in\mathbb{Z}^3} \sum_{\substack{j=1 \\ j \neq i \text{ for } \mathbf{n}=0}}^{N} \left(\frac{q_j}{\|\mathbf{x}_i - \mathbf{x}_j - \mathbf{n}\|} + \frac{\mathbf{m}_j \cdot (\mathbf{x}_i - \mathbf{x}_j - \mathbf{n})}{\|\mathbf{x}_i - \mathbf{x}_j - \mathbf{n}\|^3} \right) \qquad \text{(A.6)}$$

  for $i = 1, \ldots, N$, instead of an approximation to (A.1). This means that all given charges in the basic box (unit cube) are replicated periodically to all boxes obtained by translation of the basic box due to all possible vectors $\mathbf{n} \in \mathbb{Z}^3$. Note that the outer sum in (A.6) is only conditionally convergent. Consequently, some convention about the order of summation is necessary. This is explained in more detail in Section **??**, see also [1].

Counting Events Using PAPI

- `PAPIeventSet`

  The PAPI library[4] provides a standard interface for accessing hardware performance counters available on most modern microprocessors. If FMMV has been compiled with PAPI support enabled (, A

```
struct FmmvOptions options = fmmvGetDefaultOptions();
...
PAPI_library_init(PAPI_VER_CURRENT);
options.PAPIeventSet = PAPI_NULL;
PAPI_create_eventset(&options.PAPIeventSet);
...
PAPI_add_event(options.PAPIeventSet, PAPI_TOT_INS);
...
```

  The functions of the PAPI library have return values indicating the success of the respective operations. These have been ignored in the above code segment for simplicity. This, of course, is not recommended for actual usage of the PAPI library.

[4] PAPI is available at `http://icl.cs.utk.edu/papi/`.

A.1.5 Statistics

If a pointer to a statistics structure whose declaration is given below, is supplied to the routines of FMMV timing and memory usage information is collected during the execution of the code. This information can conveniently be printed in readable form to standard output by calling the routine

```
void printFmmvStatistics(struct FmmvStatistics *statistics);
```

The declaration of the statistics structure `FmmvStatistics` is as follows:

```
#define STAT_MAX 20
#define MAX_NUM_PAPI_EVENTS 10

struct FmmvStatistics {
    double time[STAT_MAX];
    double etime[STAT_MAX];

    int PAPIeventSet;
    long long int PAPIvalues[STAT_MAX][MAX_NUM_PAPI_EVENTS];

    unsigned int maxAllocatedMemory;

    int p;
    int s;

    int noOfParticles;
    int noOfTargets;
    int noOfSourceLevels;
    int noOfTargetLevels;

    int noOfSourceBoxes;
    int noOfTargetBoxes;
    int noOfSourceLeafBoxes;
    int noOfTargetLeafBoxes;
    float averageNoOfParticlesPerLeafBox;
    float averageNoOfTargetsPerLeafBox;

    int noOfParticlesInLevel[52];
    int noOfTargetsInLevel[52];
    int noOfSourceBoxesInLevel[52];
    int noOfTargetBoxesInLevel[52];
    int noOfSourceLeafBoxesInLevel[52];
```

```
    int noOfTargetLeafBoxesInLevel[52];
    float averageNoOfParticlesPerLeafBoxInLevel[52];
    float averageNoOfTargetsPerLeafBoxInLevel[52];
};
```

The meaning of most of the fields should be self-explanatory, so only some of them are described in the following.

- `p` and `s` are the values used for the parameters mentioned in Section A.1.4, which determine the accuracy of the FMM.

- `maxAllocatedMemory` is the maximum size of memory in bytes, allocated during the execution of the code. This value is determined by keeping track to each call of the system routines `malloc` and `free` (or to similar routines supporting aligned memory allocation). This also allows an internal check, whether the entire allocated memory has been properly deallocated.

- `time` and `etime`: These arrays provide timing information itemized to substeps of the FMM algorithm. The timing in `time` (processor time) is obtained via the system routine `clock`, whereas the value provided in `etime` (elapsed time) is obtained via the system routine `gettimeofday`. The available steps are given by the following enum constants, such that, e.g., `time[STAT_M2M]` gives the timing of the M2M substep of the FMM:

```
enum StatStep {
    STAT_TOTAL,
    STAT_BUILD_TREE,
    STAT_GEN_M,
    STAT_M2M,
    STAT_M2L,
    STAT_L2L,
    STAT_EVAL_L,
    STAT_LIST1,
    STAT_LIST3,
    STAT_LIST4,
    STAT_LIST34,
    STAT_FARFIELD,
    STAT_NEARFIELD,
    STAT_INITIALIZE,
    STAT_EVALUATE,
    STAT_FINALIZE,
    _STAT_LAST_ /* mark last entry, do not remove! */
};
```

If some substep is not applicable, the corresponding values are set to -1.

- `PAPIvalues`:

A.2 Calling FMMV from Python

A.2.1 General Remarks

FMMV provides an interface for the programming language Python with similar functionality as the C interface, cf. Section A.1. The interface depends on the package `numarray`[5], which adds efficient array manipulation capabilities to the Python programming language. Python programs calling one of the double precision routines of FMMV must import the modules `numarray` and `fmmv`:

```
from numarray import *
from fmmv import *
```

For single precision, the module `fmmvf` has to be imported instead of `fmmv`:

```
from numarray import *
from fmmvf import *
```

In either case, this gives access to the routines

```
coulomb3d
coulomb3d_initilize
coulomb3d_evaluate
coulomb3d_finalize
```

which calculate in double or single precision, depending on whether they were imported via `fmmv` or `fmmvf`.

Error Handling

In case of error the routines of the modules `fmmv` and `fmmvf` throw an exception, which—if not caught by an exception handler—causes the calling program to abort after printing an error message.

[5] `numarray` is available at `http://www.stsci.edu/resources/software_hardware/numarray`.

A.2.2 `coulomb3d`

The most basic usage of `coulomb3d` looks like

```
pot = coulomb3d(particles, charges)
```

Here, `particles` and `charges` are `numarray` arrays containing the coordinates of the particles and the corresponding charges, respectively. They are generated by, e.g.,[6]

```
particles = zeros((NParticles, 3), type=Float64)
charges = zeros(NParticles, type=Float64)
```

where `NParticles` denotes the number of particles, and `type` indicates the type of arrays to be generated (`Float32` for single and `Float64` for double precision), and have then to be filled with their intended values by some Python code.

After successful execution of `coulomb3d`, `pot` is an array containing an approximation to the potential (A.1) without dipole sources, i.e., $\mathbf{m}_i = 0$.

Dipole Sources

The usage of `coulomb3d` for dipole sources, whose dipole moments are supplied by the array `dipoleMoments`, looks like

```
pot = coulomb3d(particles, dipoleMoments=dipoleMoments)
```

or

```
pot = coulomb3d(particles, charges, dipoleMoments=dipoleMoments)
```

where in the latter case the particles also carry monopole charges.

Computing Gradients

The additional argument `computeGradient=True` requires `coulomb3d` to compute approximations to the gradients (A.2):

```
(pot, grad) = coulomb3d(..., computeGradient=True)
```

Here and in the following the dots indicate further required arguments, e.g., the ones described above.

[6] Notice the order of the dimensions `(NParticles, 3)` in the definition of `particles` (and similarly for `dipoleMoments`). This resembles the fact that in Python arrays are stored *row-wise*. The j-th coordinate of the i-th particle is accessed by `particles[`i`,`j`]`, where $j \in \{0, 1, 2\}$ and $i \in \{0, \ldots,$ `nparticles`$-1\}$.

Independent Target Locations

If the potential and optionally the gradient shall be computed at independent target locations, the Cartesian coordinates of these locations have to be supplied by an appropriate array `targets`:

```
pot = coulomb3d(..., targets=targets)
```

or

```
(pot, grad) = coulomb3d(..., targets=targets, computeGradient=True)
```

Further Options

For the fine-tuning of the algorithm `coulomb3d` supports the following options, whose meanings are exactly the same as for the corresponding options of the C interface of FMMV described in Section A.1.4:

```
precision
useApproxInvSqrt
splitThreshold
maxLevel
splitTargetThreshold
maxTargetLevel
useHilbertOrder
scale
useFarfieldNearfieldThreads
periodicBoundaryConditions
```

Options are set according to the scheme

```
fmmvHandle = coulomb3d(..., option1=value1, option2=value2,...)
```

For missing options, or for options explicitly set to the value `None`, default values as described in Section A.1.4 are taken.

Statistics

The additional argument `getStatistics=True` requires `coulomb3d` to collect timing and memory usage information:

```
(pot, stat) = coulomb3d(..., getStatistics=True)
```

or

```
(pot, grad, stat) = coulomb3d(..., computeGradient=True, getStatistics=True)
```

Here, the output parameter `stat` is a dictionary with the following strings as keys:

```
"p"
"s"
"maxAllocatedMemory"
"noOfParticles"
"noOfTargets"
"noOfSourceLevels"
"noOfTargetLevels"
"noOfSourceBoxes"
"noOfTargetBoxes"
"noOfSourceLeafBoxes"
"noOfTargetLeafBoxes"
"averageNoOfParticlesPerLeafBox"
"averageNoOfTargetsPerLeafBox"
"noOfParticlesInLevel"
"noOfTargetsInLevel"
"noOfSourceBoxesInLevel"
"noOfTargetBoxesInLevel"
"noOfSourceLeafBoxesInLevel"
"noOfTargetLeafBoxesInLevel"
"averageNoOfParticlesPerLeafBoxInLevel"
"averageNoOfTargetsPerLeafBoxInLevel"
"totalTime"
"buildTreeTime"
"genMTime"
"M2MTime"
"M2LTime"
"L2LTime"
"evalLTime"
"list1Time"
"list3Time"
"list4Time"
"list34Time"
"farfieldTime"
"nearfieldTime"
"initializeTime"
"evaluateTime"
"finalizeTime"
```

The meanings of the keys are self-explanatory or explained in Section A.1.5. Keys ending with `"Time"` provide the running times of the corresponding substeps of the FMM obtained via the system routine `gettimeofday`, or are missing in the dictionary `stat` if the corresponding substeps are not applicable. For example, `stat["M2MTime"]` gives the running time of the M2M substep of the FMM, and `stat["noOfSourceBoxesInLevel"][3]` gives the number of source boxes in level 3 of the oct-tree data structure.

A.2.3 `coulomb3d_initialize`, `coulomb3d_evaluate`, `coulomb3d_finalize`

Analogously to the C interface described in Section A.1.3, the Python interface of FMMV provides separate routines

```
coulomb3d_initialize
coulomb3d_evaluate
coulomb3d_finalize
```

for the three main steps—initialization, evaluation, and finalization—of the algorithm. Now, the basic usage looks like

```
fmmvHandle = coulomb3d_initialize(particles)
...
pot = coulomb3d_evaluate(fmmvHandle, charges)
...
coulomb3d_finalize(fmmvHandle)
```

The usage is similar to the usage of `coulomb3d` (cf. Section A.2.2), but with the arguments appropriately distributed among the routines `coulomb3d_initialize` and `coulomb3d_evaluate`, and with a handle `fmmvHandle` serving for the internal communication between the routines.

Dipole Sources

The kind of sources to be handled can be determined by the argument `typeSources` of `coulomb3d_initialize`:

- `typeSources=0` for monopole charges only. As shown above, this does not have to be explicitly specified.
- `typeSources=1` for dipole charges only:

```
fmmvHandle = coulomb3d_initialize(particles, typeSources=1)
...
pot = coulomb3d_evaluate(fmmvHandle, dipoleMoments=dipoleMoments)
...
```

- `typeSources=2` for monopole and dipole charges:

```
fmmvHandle = coulomb3d_initialize(particles, typeSources=2)
...
pot = coulomb3d_evaluate(fmmvHandle, charges, dipoleMoments=dipoleMoments
...
```

In the latter two cases the argument `dipoleMoments` of `coulomb3d_evaluate` is of course mandatory.

Computing Gradients

The argument `computeGradient=True` of `coulomb3d_initialize` requires `coulomb3d_evaluate` to compute approximations to the gradients (A.2):

```
fmmvHandle = coulomb3d_initialize(..., computeGradient=True)
...
(pot, grad) = coulomb3d_evaluate(fmmvHandle, ...)
...
```

Independent Target Locations

Independent target locations supplied to `coulomb3d_initialize`,

```
fmmvHandle = coulomb3d_initialize(..., targets=targets)
```

cause subsequent calls of `coulomb3d_evaluate` to calculate their results at these given locations.

Further Options

Options can be set at time of initialization only:

```
fmmvHandle = coulomb3d_initialize(..., option1=value1, option2=value2,...)
```

These options then also apply to subsequent calls of `coulomb3d_evaluate`.

Statistics

If `getStatistics=True` is specified for `coulomb3d_initialize`, then timing and memory usage information is collected during execution of *all* of the three steps initialization, evaluation, and finalization:

```
(fmmvHandle, stat) = coulomb3d(..., getStatistics=True)
...
(pot, stat) = coulomb3d_evaluate(fmmvHandle, ...)
# or possibly (pot, grad, stat) = ...
...
stat = coulomb3d_finalize(fmmvHandle, ...)
```

A.3 Calling FMMV from MATLAB

A.3.1 General Remarks

FMMV provides an interface for easy access of the library from within MATLAB. This interface has been modeled after the Python interface described in Section A.2 as far as it is possible and compatible with the conventions used in MATLAB. Consequently, also the documentation of the MATLAB interface given in this section follows closely Section A.2.

Currently the MATLAB interface of FMMV provides only routines using double precision, which are the following:

```
fmmvcoulomb3d
fmmvcoulomb3d_initialize
fmmvcoulomb3d_evaluate
fmmvcoulomb3d_finalize
```

These routines are readily available as soon as their corresponding program files (shared libraries) are located in some directory on the MATLAB path. No explicit initialization of the library like importing some modules is necessary.

Error Handling

In case of error the routines of the MATLAB interface of FMMV throw an exception, which—if not caught by an exception handler— causes the calling program to stop execution after printing an error message.

A.3.2 `fmmvcoulomb3d`

The basic usage of `fmmvcoulomb3d` is

```
pot = fmmvcoulomb3d(particles, charges)
```

where the arrays `particles` and `charges` contain the coordinates of the particles and the corresponding charges, respectively. These arrays are generated by, e.g.,

```
particles = zeros(3, nparticles)
charges = zeros(1, nparticles)
```

where `nparticles` denotes the number of particles.[7]

Dipole Sources

For dipole sources with dipole moments given by the array `dipoleMoments` the usage of `fmmvcoulomb3d` is

```
pot = fmmvcoulomb3d(particles, [], dipoleMoments)
```

or

```
pot = fmmvcoulomb3d(particles, charges, dipoleMoments)
```

where in the first case the empty matrix `[]` serves as a placeholder for the absent monopole charges, and in the latter case the particles also carry monopole charges.

Computing Gradients

Gradients are computed, if a second output parameter `grad` is supplied to the call of the routine `fmmvcoulomb3d`:

```
[pot, grad] = fmmvcoulomb3d(...)
```

[7] Notice the order of the dimensions of the array `particles`, which resembles the fact that in MATLAB arrays are stored *column-wise*. The j-th coordinate of the i-th particle is accessed by `particles(`j`,`i`)`, where $j \in \{1, 2, 3\}$ and $i \in \{1, \ldots,$ `nparticles`$\}$.

Independent Target Locations

Independent target locations can be supplied by a fourth argument `targets`:

```
pot = fmmvcoulomb3d(particles, charges, dipolemoments, targets)
```

or

```
[pot, grad] = fmmvcoulomb3d(particles, charges, dipolemoments, targets)
```

Here the potentials and gradients are computed at the locations given by the array `targets`, not at those locations given by the array `particles`.

Further Options

For the fine-tuning of the algorithm, an options structure can be supplied to `fmmvcoulomb3d` as a fifth argument:

```
[pot, grad] = fmmvcoulomb3d(particles, charges, dipolemoments, targets, optior
```

For the creation of such options structures FMMV provides the MATLAB function `fmmvset` with the following syntax:[8]

```
options = fmmvset('option1', value1, 'option2', value2, ...)
```

Here `option1`, `option2`,... must be one of the following options, whose meanings are exactly the same as for the corresponding options of the C interface of FMMV described in Section A.1.4:

```
precision
useApproxInvSqrt
splitThreshold
maxLevel
splitTargetThreshold
maxTargetLevel
useHilbertOrder
scale
useFarfieldNearfieldThreads
periodicBoundaryConditions
printStatistics
```

Any unspecified options have default values as described in Section A.1.4.

[8] This method of handling options is modeled after the one used by the ODE solvers of MATLAB.

Statistics

One of the options described above is `printStatistics`. If this is set to a non-zero value, timing and memory usage information collected during the execution of the code is printed in readable form to standard output.

A.3.3 `fmmvcoulomb3d_initialize`, `fmmvcoulomb3d_evaluate`, `fmmvcoulomb3d_finalize`

Analogously to the C interface described in Section A.1.3, the MATLAB interface of FMMV provides separate routines

```
fmmvcoulomb3d_initialize
fmmvcoulomb3d_evaluate
fmmvcoulomb3d_finalize
```

for the three main steps—initialization, evaluation, and finalization—of the algorithm. The basic usage now looks like

```
fmmvhandle = fmmvcoulomb3d_initialize(particles)
...
pot = fmmvcoulomb3d_evaluate(fmmvhandle, charges)
...
fmmvcoulomb3d_finalize(fmmvhandle)
```

Here, `fmmvHandle` serves for the internal communication between the routines.

Dipole Sources

The kind of sources to be handled can be determined by a second argument `type` to `fmmvcoulomb3d_initialize`:

- `type=0` for monopole charges only. As shown above, this is the default and does not have to be explicitly specified.
- `type=1` for dipole charges only:

  ```
  fmmvhandle = fmmvcoulomb3d_initialize(particles, 1)
  ...
  pot = fmmvcoulomb3d_evaluate(fmmvhandle, [], dipolemoments)
  ...
  ```

In this case the second argument of `fmmvcoulomb3d_evaluate` is ignored and can thus be, e.g., the empty matrix `[]`.

- `type=2` for monopole and dipole charges:

```
fmmvhandle = fmmvcoulomb3d_initialize(particles, 2)
...
pot = fmmvcoulomb3d_evaluate(fmmvhandle, charges, dipolemoments)
...
```

In the latter two cases the argument `dipolemoments` of `fmmvcoulomb3d_evaluate` is of course mandatory.

Computing Gradients

A third non-zero argument of `fmmvcoulomb3d_initialize` requires `fmmvcoulomb3d_evaluate` to compute gradients.

```
fmmvhandle = fmmvcoulomb3d_initialize(charges, type, 1)
...
[pot, grad] = fmmvcoulomb3d_evaluate(fmmvhandle, charges, dipolemoments)
...
```

In this case, the second output parameter `grad` of `fmmvcoulomb3d_evaluate` is mandatory.

Independent Target Locations

Independent target locations supplied to `fmmvcoulomb3d_initialize`,

```
fmmvhandle = fmmvcoulomb3d_initialize(particles, type, grad, targets)
```

cause subsequent calls of `fmmvcoulomb3d_evaluate` to calculate their results at these given locations.

Further Options

An options structure can be supplied at time of initialization only:

```
fmmvhandle = fmmvcoulomb3d_initialize(particles, type, grad, targets, options)
```

The corresponding options then also apply to subsequent calls of `fmmvcoulomb3d_evaluate`.

References

[1] M. Challacombe, C. A. White, M. Head-Gordon, *Periodic boundary conditions and the fast multipole method*, J. Chem. Phys. **107**, 23 (1997), pp. 10131–10140.

[2] C. A. White, M. Head-Gordon, *Fractional tiers in fast multipole calculations*, Chem. Phys. Letters 257 (1996), pp. 647–650.