ScaFaCoS Manual

Matthias Bolten
Department of Mathematics and Science, University of Wuppertal
bolten@math.uni-wuppertal.de

Florian Fahrenberger
Institute for Computational Physics, University of Stuttgart

René Halver
Institute for Advanced Simulation, Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH

Frederik Heber
Institute for Numerical Simulation, University of Bonn

Michael Hofmann
Faculty of Computer Science, Technische Universität Chemnitz

Ivo Kabadshow
Institute for Advanced Simulation, Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH

Olaf Lenz
Institute for Computational Physics, University of Stuttgart

Michael Pippig
Faculty of Mathematics, Technische Universität Chemnitz
michael.pippig@mathematik.tu-chemnitz.de
http://www.tu-chemnitz.de/~mpip

Mathias Winkel
Institute for Advanced Simulation, Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH
pepc@fz-juelich.de
http://www.fz-juelich.de/ias/jsc/pepc

August 8, 2018

Contents

I. User's Guide 9

1. Introduction 11
  1.1. What ScaFaCoS Computes 11
  1.2. Acknowledgements 12
  1.3. Licensing 12
  1.4. Requirements 12
  1.5. Feature Matrix 13

2. Compiling and installing ScaFaCoS 15

3. Interface 19
  3.1. Basic Ideas of the Interface 19
  3.2. Use of the ScaFaCoS Library 20
  3.3. Error Handling 27
  3.4. Fortran Specifics 28
  3.5. Further Functionality 28

4. FMM – Fast Multipole Method 31

5. MEMD – Maxwell Equation Molecular Dynamics 33
  5.1. Description of the method 33
  5.2. Systems suited for the algorithm 35
  5.3. Solver-specific parameters 36
  5.4. Solver-specific functions 37
  5.5. Known bugs or missing features 38

6. MMM1D 39

7. MMM2D 41

8. Ewald 43

9. P^2NFFT– Particle-Particle NFFT 45
  9.1. Description of the Method 45
  9.2. Features 47
  9.3. Solver-specific Parameters 47
  9.4. Solver-specific Functions 54

10. P3M – Particle-Particle Particle-Mesh Ewald 61
- 10.1. Features . . . 61
- 10.2. Solver-specific Parameters . . . 61

11. PEPC – Pretty Efficient Parallel Coulomb Solver 63

12. PP3MG – NameExpanded 69

13. vmg – Versatile Multigrid 71

14. direct – Direct summation 77

15. List of Functions 79
- 15.1. Mandatory Functions . . . 79
- 15.2. Generic Functions . . . 82
- 15.3. Direct Solver specific Functions . . . 87
- 15.4. Ewald Solver specific Functions . . . 87
- 15.5. FMM Solver specific Functions . . . 87
- 15.6. MEMD Solver specific Functions . . . 87
- 15.7. MMM1D Solver specific Functions . . . 87
- 15.8. MMM2D Solver specific Functions . . . 87
- 15.9. PEPC Solver specific Functions . . . 87
- 15.10PP3MG Solver specific Functions . . . 87
- 15.11P^2NFFT Solver specific Functions . . . 87
- 15.12P^3M Solver specific Functions . . . 87
- 15.13VMG Solver specific Functions . . . 90

II. Developer's Guide 91

16. Build System 93

17. Implementation 97
- 17.1. Licenses . . . 97

18. Test Environment 99
- 18.1. Generic Test Program `scafacos_test` . . . 99
- 18.2. Numerical Results . . . 104
- 18.3. Generation of Simulation Data . . . 104
- 18.4. Error . . . 104
- 18.5. Distributions and Results . . . 105

III. Bibliography 109

19. Bibliography 111

IV. Index 113

Index 119

Todo list

Fix introduction . . . 11
Virial . . . 11
Virial . . . 12
Is MPI 1.2 really enough? . . . 13
FFTW3 is part of the library . . . 13
Should the near field solver be available to the user? . . . 28
quantify these requirements . . . 67
warn only once on one process, if box is non-cubic . . . 67
move PEPC citations to bibliography . . . 67
move vmg citations to bibliography . . . 75
fcs_method_has_near fehlt! . . . 83

Part I.

User's Guide

1. Introduction

Fix introduction

1.1. What ScaFaCoS Computes

1.1.1. Fully Non-Periodic Boundary Conditions

In this part we apply our fast summation algorithm to a system of M charged particles located at source nodes $\mathbf{x}_l \in \mathbb{R}^3$ with charge $q_l \in \mathbb{R}$. We are interested in the evaluation of the electrostatic potential ϕ at target node $\mathbf{x}_j \in \mathbb{R}^3$,

$$\phi(\mathbf{x}_j) = \sum_{\substack{l=1\\ l\neq j}}^{M} \frac{q_l}{\|\mathbf{x}_j - \mathbf{x}_l\|_2}, \quad j = 1, \dots, M, \tag{1.1}$$

and the electrostatic fields $\mathbf{E}$ evaluated at position $\mathbf{x}_j \in \mathbb{R}^3$,

$$\mathbf{E}(\mathbf{x}_j) = -\sum_{\substack{l=1\\ l\neq j}}^{M} q_l \frac{\mathbf{x}_j - \mathbf{x}_l}{\|\mathbf{x}_j - \mathbf{x}_l\|_2^3}, \quad j = 1, \dots, M. \tag{1.2}$$

Furthermore we are interested in the computation of the total electrostatic potential energy

$$U := \frac{1}{2} \sum_{j=1}^{M} q_j \phi(\mathbf{x}_j),$$

which can be evaluated straightforward after the computation of the potentials $\phi(\mathbf{x}_j), j = 1, \dots, M$.

Virial

1.1.2. Fully Periodic Systems

We consider a system of charged particles coupled via the Coulomb potential, a cubic simulation box with edge length B, containing M charged particles, each with a charge $q_l \in \mathbb{R}$, located at $\mathbf{x}_l \in [0, B)^3$. Periodic boundary conditions in a system without cut-off is represented by replicating the simulation box in all directions of space. The electrostatic potential ϕ at $\mathbf{y} \in [0, B)^3$, can be written as a lattice sum, see [6, Chapter

12] and [12],

$$\phi(\mathbf{x}_j) = \sum_{\mathbf{r}\in\mathbb{Z}^3} \sum_{\substack{l=1 \\ l\neq j \text{ for } \mathbf{r}=\mathbf{0}}}^{M} \frac{q_l}{\|\mathbf{x}_j - \mathbf{x}_l + \mathbf{r}B\|_2}, \quad j = 1,\dots,M \tag{1.3}$$

and the electrostatic field $\mathbf{E}$ evaluated at position $\mathbf{x}_j \in [0,B)^3$ is given by

$$\mathbf{E}(\mathbf{x}_j) = -\sum_{\mathbf{r}\in\mathbb{Z}^3} \sum_{\substack{l=1 \\ l\neq j \text{ for } \mathbf{r}=\mathbf{0}}}^{M} q_l \frac{\mathbf{x}_j - \mathbf{x}_l + \mathbf{r}B}{\|\mathbf{x}_j - \mathbf{x}_l + \mathbf{r}B\|_2^3}, \quad j = 1,\dots,M\,, \tag{1.4}$$

Furthermore we are interested in the computation of the total electrostatic potential energy

$$U := \frac{1}{2}\sum_{j=1}^{M} q_j \phi(\mathbf{x}_j),$$

which can be evaluated straightforward after the computation of the potentials $\phi(\mathbf{x}_j)$, $j = 1,\dots,M$ like in the non-periodic case.

Virial

1.2. Acknowledgements

This is a network project of German research groups working on a unified parallel library for various methods to solve electrostatic (and gravitational) problems in large particle simulations. The project is financed by the German Ministry of Education and Science (BMBF) under contract number 01 IH 08001 A-D. Main focus of the project is to provide methods for electrostatic problems and to implement efficient parallel methods in order to scale up to thousands of processors. ScaFaCoS is supported within the period 01/01/2009 - 31/12/2011.

1.3. Licensing

This library is open source software. It comes with parts that are distributed under the GNU General Public License (GPL) and parts that are distributed under the GNU Lesser General Public License (LGPL). Please refer to the documentation of the individual solver methods for details. The toplevel configure script provides a switch `--disable-gpl` to disable the build of all methods which allow only a distribution under the GPL.

1.4. Requirements

The following libraries and tools are required to be able to compile and use ScaFaCoS:

MPI As ScaFaCoS is parallel software, you will need a working MPI environment that implements at least the MPI standard version 1.2.

PI 1.2 re-
nough?

C99 compiler The C-code of ScaFaCoS uses the C99 standard, therefore a compiler that is able to compile C99 code must be available. We do not know of any recent platform that does not provide a C99 compiler.

C++ compiler Some methods (*e.g.* VMG) require a C++ compiler.

Fortran 2003 compiler The Fortran code of some methods (*e.g.* FMM) does require a Fortran 2003 compiler.

FFTW Some solver methods (*e.g.* P^{3}M and P^{2}NFFT) need the FFTW library version 3.3.0 or later [1] for Fourier transforms. For P^{2}NFFT, the MPI interface of FFTW3 has to be available. You do not only need the library itself, but also the header files. Depending on the operating system, these may come in separate development packages (*e.g.* `fftw3-dev`).

FFTW3 is part
of the library

1.5. Feature Matrix

	FMM	MEMD	MMM*D	P^{2}NFFT	P^{3}M	PEPC	PP3MG	VMG	Direct	Ewald
3D-periodic	+	+	−	+	+	+	+	+	+	+
non-cubic, cuboid	−	−		+	+	+[a]	+	?	+	+
non-cubic, non-cuboid	?	?		+	+	+[a]	?	?	+	?
Virials	?	−		?[b]	−[c]	−[c]	(scalar)	?	+	−
Nonperiodic	+	−	−	+	−	+	−	?	+	−
non-cubic	−			+		+		?	+	
Virials	+			+		−[c]		?	+	
Partially periodic	+	−	+	+	−	+[a]	−	?	+	−
Tunable accuracy	+	−	+	+	+	−	−	−	−	+
Delegate near-field	−	−	−	+	+	−	?	?	−	−

[a] untested
[b] diagonal approximation
[c] implementation planned

Table 1.1.: Overview of the features of the different solvers.

---

[1] http://www.fftw.org/

2. Compiling and installing ScaFaCoS

The ScaFaCoS library uses the GNU build system and thus compiling and installing consists of the common steps `configure` and `make`. In the following, the steps for building the library from source are explained in more details.

1) Obtain a copy of the ScaFaCoS library sources. Single releases of the library sources can be downloaded from the official web site (`http://www.scafacos.de`). Furthermore, the sources of the library are hosted on GitHub where the lasted version is publicly available (see `http://github.com/scafacos`). A local copy of the library repository can be obtain with the following command:

   ```
   git clone git@github.com:scafacos/scafacos.git
   ```

   Building the library from the git repository requires recent versions of the GNU Autotools, *i.e.* m4, Autoconf, Automake, Libtool. Details about the specific versions required can be found in Sect. 16. Executing Autotools has to be performed with the `bootstrap` script:

   ```
   ./bootstrap
   ```

   After executing this script, the source directory should contain a newly generated `configure` script.

2) Compiling the library consists of the two steps `configure` and `make`. For the following description it is assumed that the library sources are stored in directory `scafacos/`.

   a) The library can be build either inside or outside the source tree. To build the library outside the source tree, change the working directory to the desired build directory:

      ```
      mkdir ../build_fcs
      cd ../build_fcs
      ```

      Execute the `configure` script as follows:

      ```
      ../scafacos/configure <options>
      ```

      See '`./configure --help`' (or '`./configure --help=recursive`') to display all supported options. The main options of the library are listed in Table 2.1.

   b) Run `make` to build the library. Use option `-j <N>` to execute (at most) `N` build jobs in parallel.

3) Run test programs:
   a) Run `make check` to execute all test programs of the library. The build system tries to finding out how MPI jobs can be started. Edit file `test/defs.in` to modify the method that was automatically determined. Alternatively, `configure` option `MPIEXEC=...` can be used to define the command for executing MPI programs.
   b) The test programs are located in `test/c` and `test/fortran`. See scripts `start_....sh` to execute single test programs.
   c) If the execution of MPI programs with the previous methods fails, then it is also possible to start the test programs manually. Use scripts `start_....sh` (in `test/c/` or `test/fortran/`) and replace command `start_mpi_job` with an appropriate command for executing MPI programs.

4) Run `make install` to install the header and library files (*i.e.*, to `/usr/local/` if not specified otherwise with `--prefix=...` during `configure`). Alternatively, the header and library files can be found in `src/` and `lib/`, respectively.

5) A configuration file `scafacos.pc` for `pkg-config` is created in `packages/` and installed with `make install`. Use the following commands to determine the compiler and linker flags required to integrate the library into an application program:

```
pkg-config --cflags scafacos
pkg-config --libs scafacos
```

`-C`	Enable caching to speed up `configure`.
`--prefix=PREFIX`	Install library files to directory `PREFIX`.
`--enable-fcs-solvers=...`	Enable only the specified list of solvers.
`--disable-fcs-SOLVER`	Disable the specified solver `SOLVER`.
`--enable-fcs-int=...`	Set ScaFaCoS integer type to the given C type.
`--enable-fcs-float=...`	Set ScaFaCoS float type to the given C type.
`--enable-fcs-integer=...`	Set ScaFaCoS integer type to the given Fortran type.
`--enable-fcs-real=...`	Set ScaFaCoS float type to the given Fortran type.
`--enable-fcs-info`	Enable info output.
`--enable-fcs-debug`	Enable debug output.
`--enable-fcs-timing`	Enable timing output.
`--enable-single-lib`	Build and install only a single file `libfcs.a`.
`MPICC=...  CFLAGS=...`	Set (MPI) C compiler and flags.
`MPICXX=...  CXXFLAGS=...`	Set (MPI) C++ compiler.
`MPIFC=...  FCFLAGS=...`	Set (MPI) Fortran compiler.

Table 2.1.: List of common `configure` options of the ScaFaCoS library.

3. Interface

3.1. Basic Ideas of the Interface

The basic idea of the ScaFaCoS interface is that it should be very simple to include the library to existing programs and packages. Therefore the aim was to have as few required functions as possible for adding the library. To use the library one has only five functions to add to an existing code, if the default parameters for each solver suffice. Of course there is the possibility to change the parameters of the solvers by use of additional functions. In this sections the basic functions will be described, so that the reader is able to add the ScaFaCoS library to his or her own code.

To understand the following sections, one has to know the idea behind the use of the library. One usage cycle is divided into five sections, which are mapped to the five required functions. These sections are:

1. Initialization of the library
2. Setting the non-solver specific parameters
3. Tuning the solver
4. Running the solver (more then once if needed)
5. Destroying allocated resources of the library

These five steps are each tied to one call of the respective function. In the initialization step the communicator used by the library is defined, as well as the solver to used is chosen. During the second step the global parameters like system size and periodicity of the system are set. Within the third step solver-specific tuning methods are called (where possible) which try to utilize the system in order to improve the solvers performance or set up data structures requiring knowledge about particle positions. The central step is the fourth step in which the calculation of the Coulomb interactions takes place. E.g. within a MD code this step can be called for each time step, in order to take into consideration the shifting particle positions. After the calculation is or calculations are done, the last step frees all the resources which were used by the library.

The interface functions are provided in a (C++ compatible) C version and in a Fortran version, which is a wrapper of the C version. In the section both variants will be described. Table 3.1 gives an overview about all solvers included until the creation of this documentation.

ScaFaCoS abbreviation	full solver name
direct	Direct solver (14)
ewald	Ewald summation (8)
fmm	Fast Multipole Method (4)
memd	Maxwell equations Molecular Dynamics (5)
mmm1d	(6)
mmm2d	(7)
pepc	Pretty Efficient Parallel Coulomb solver (Barnes-Hut Tree Code) (11)
pp3mg	Particle-Particle Particle-Multigrid solver (12)
p2nfft	Parallel Particle Non-Equidistant FFT solver (9)
p3m	Particle-Particle Particle-Mesh method (10)
vmg	Versatile Multigrid method (13)

Table 3.1.: Solvers implemented in ScaFaCoS (02/2012)

3.2. Use of the ScaFaCoS Library

In this section the implementation of the steps presented in the previous section will be explained. Before the actual implementation of the ScaFaCoS library to a code will be described, the headers and modules and the data types and constants provided by the library will be presented.

3.2.1. Header and Modules

To simplify the inclusion of the library into existing or newly written codes, the library provides either a single header for C or a single module for Fortran which needs to be included into the code. For C this is the *fcs.h* header and for Fortran this is the module *fcs_module* (fig. 3.1).

Due to the use of the pkg-package tool [2], there is a easy way to get the correct library and include options for the compilation of the code with ScaFaCoS included. Figure 3.2 shows how pkg-config can be used to that end at an example. In order to use pkg-config *make check* has to be used before so that an installation is created.

```
/* inclusion in C */
#include ”fcs.h”
```

```
! inclusion in Fortran
use fcs_module
```

Figure 3.1.: Use of the provided header / module of the ScaFaCoS library.

```
export PKG_CONFIG_PATH=<ScaFaCoS install path>/lib/pkg-config↩
    :${PKG_CONFIG_PATH}
FCSLIB=$(shell pkg-config --libs scafacos)
FCSINC=$(shell pkg-config --cflags scafacos)
gcc -o <application name> [other options] ${FCSLIB} ${FCSINC}↩
    <source(s)>
```

Figure 3.2.: exemplary usage of pkg-config to get the necessary information for compilation

3.2.2. Data Types and Constants

The library provides data types for C (table 3.2) and data kinds Fortran (table 3.3), which are either defined within the configure script or set to default values by the configure script. During this procedure the Fortran data types are matched to the corresponding C data types in order to ensure compatibility between the Fortran wrapper and the C functions underneath. The default values for these data types are:

In addition to that the library supplies an additional data type, which is used for data transport between the different functions. This data type is FCS, which is a pointer to a struct containing the common information about the simulation system (box size, periodicity, ...). From the user side it is only used as an argument to the different library functions and must not be changed by him. Otherwise the behavior of the library is undefined.

Furthermore there are some definitions, which are used within the library and which can

ScaFaCoS data type	data type (C)
fcs_int	int
fcs_float	double

Table 3.2.: ScaFaCoS data types for C

ScaFaCoS data type	data type (Fortran 2003)
fcs_integer_kind_isoc	C_INT
fcs_real_kind_isoc	C_DOUBLE

Table 3.3.: ScaFaCoS data kinds for Fortran

be used by the user.

The mathematical constants (table 3.4) can be used in C and in Fortran. Due to the *FCS_* prefix it is ensured, that they don't interfere with eventual definitions by the user or the system for these values. As a mean to simplify error handling within the library the error values are also defined as macros (table 3.5).

Furthermore there are two structures defined, which are used internally to store provided parameters and error details. These structures are called *FCS* and *FCSResult*. *FCS* will be created during the initialization step (3.2.3) and filled during the parameter setup step (3.2.4). After that it needs to be supplied for the tuning (3.2.5) and calculation step (3.2.6). Finally it will be freed during the clean up step (3.2.7). *FCSResult* will be used for error handling and the needed getter-functions will be explained in the section on error handling (3.3)

3.2.3. Initialization Step

During the initialization step, the library is supplied with all information that cannot be changed during the simulation by use of the function *fcs_init* (fig. 15.1). This information includes the MPI communicator, which is to be used and the chosen method. With this information a *FCS* data structure is created. The calls in C and Fortran differ only in the data types of the arguments. While C supplies an unique MPI_Comm type for MPI communicators, in Fortran these are represented by integer values. Also Fortran does not grant access to pointers in the same fashion as C does. In order to circumvent this, the Fortran interface uses the c_ptr type ([3, p. 565f.]) as a replacement.

The initialization step is mandatory for the use of the ScaFaCoS library, since in it the most important information is set. This is on the one hand the choice of solver and on the other hand the parallel environment in which the library is used, as described by the MPI communicator. (see also table 3.6)
If the function returns successfully, it returns a *NULL* value, else it returns as an error value *FCS_NULL_ARGUMENT* if one of the provided arguments is not defined or *FCS_ALLOC_FAILED* if the *FCS* structure could not be allocated. If an invalid solver was chosen, then the function returns *FCS_WRONG_ARGUMENT*, e.g. if a method was chosen, that was disabled in the configure script.

C macro name	value
FCS_E	Euler constant $e = 2.71828\ldots$
FCS_LOG2E	$\log_2 e = 1.44269\ldots$
FCS_LOG10E	$\log_{10} e = 0.43429\ldots$
FCS_LN2	$\log_e 2 = 0.69314\ldots$
FCS_LN10	$\log_e 10 = 2.30258\ldots$
FCS_PI	$\pi = 3.14159\ldots$
FCS_PI_2	$\pi/2 = 1.57079\ldots$
FCS_PI_4	$\pi/4 = 0.78539\ldots$
FCS_1_PI	$1/\pi = 0.31830\ldots$
FCS_2_PI	$2/\pi = 0.63661\ldots$
FCS_2_SQRTPI	$2/\sqrt{\pi} = 1.12837\ldots$
FCS_SQRT2	$\sqrt{2} = 1.41421\ldots$
FCS_SQRT1_2	$1/\sqrt{2} = 0.70710\ldots$

Table 3.4.: ScaFaCoS mathematical macros

3.2.4. Parameter Setup Step

This step is two-fold, the parameters describing the system are set, as well as the parameters for the solver chosen in the initialization step (3.2.3). The former is mandatory, because the information about the system is needed by the solvers in order to calculate the correct forces, while the latter is optional as the solvers provide (non-optimal) default parameters for themselves.

As can be seen in figure 15.2 the calls differ for C and Fortran. Since plain C does not provide a logical data type, e.g. bool, the C interface uses integer variables for this purpose. The Fortran data kinds are linked to the C data types set in the configure script. Possible return values are *NULL* for a successful return and either *FCS_WRONG_ARGUMENT* if an argument is not defined or *FCS_WRONG_VALUE* if an argument has an invalid value.

C macro name	meaning
FCS_SUCCESS	no error occurred (deprecated)
FCS_NULL_ARGUMENT	an argument passed to the method was undefined
FCS_ALLOC_FAILED	an internal allocation of memory failed
FCS_WRONG_ARGUMENT	it was tried to set an parameter belonging to another method
FCS_MISSING_ELEMENT	a system parameter was not set
FCS_LOGICAL_ERROR	a violation of set parameters was encountered
FCS_INCOMPATIBLE_METHOD	the method cannot be used for given system
FCS_MPI_ERROR	an MPI error was encountered
FCS_FORTRAN_CALL_ERROR	an error occurred while using the Fortran wrapper

Table 3.5.: ScaFaCoS error values

parameter name	description	valid values
handle	pointer to parameter containing structure	FCS (C) or type (c_ptr) (Fortran)
method	chosen method	abbreviation from table 3.1
comm	MPI communicator the library has to work on	valid MPI communicator

Table 3.6.: Parameters for fcs_init

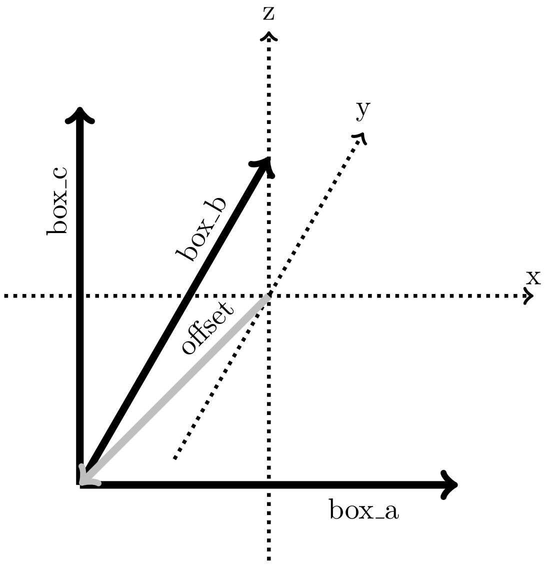

Figure 3.3.: Scheme of system parameters describing the form and position of the simulation box

The common setup requires basic data about the simulation system. An overview can be seen in table 3.7. The box vectors describe the form of the box, in which the particles reside. For each solver there can be certain restrictions as for which kind of systems they can simulate. To get an idea which solvers work best (or work at all) with which systems please refer to the corresponding solver description within this documentation. In general no solver is able to handle non-orthogonal boxes as of yet, although the interface should be able to handle these kind of boxes. The offset enables the library to handle systems, which are shifted from the coordinate origin. For a graphical scheme, please refer to figure 3.3. The use of periodicity is likewise restricted as the box forms. Not every solver is able to cope with every combination of periodicity. The only common parameter not related to the system is the near field flag, which determines whether a method should perform its near field computations by itself or not. By default, all solvers compute the interactions entirely by themselves. However, some solvers provide the possibility to delegate their near field computations to the main program. If the near field flag is set to false (i.e. 0), then interactions inside a solver-specific cutoff range are not computed. The cutoff range can be retrieved (set) with a separate getter (setter) function. The solver methods provide separate potential functions for performing their near field computations in the main program. The functionality of the near field flag is currently supported by methods P^{3}M and P^{2}NFFT.

To change the solver-specific parameters, there are solver-specific setter functions. These are called *fcs_<solver>_set_<parameter>* where *<solver>* is the abbreviation of the solver (table 3.1) and parameter the name of the relevant parameter. If one wishes

parameter name	description	valid values
handle	pointer to parameter containing structure	FCS (C) or type (c_ptr) (Fortran)
near_field_flag	leave the near field calculations to the library	true ($\neq 0$) / false (0)
box_a	first vector describing the system box	$\in \mathbb{R}^3$
box_b	second vector describing the system box	$\in \mathbb{R}^3$
box_c	third vector describing the system box	$\in \mathbb{R}^3$
offset	offset of the lower front left corner from $\mathbf{0}$	$\in \mathbb{R}^3$
periodicity	periodicity of the system in each dimension (x,y,z)	true / false
total_particles	total amount of particles in the system	$\in \mathbb{N}^+$

Table 3.7.: Parameters for fcs_common_set

to change all parameters of a solver, the use of *fcs_<solver>_setup* is advised. A list of solver-specific parameters, that can be changed and possible values are available in the chapters describing the solvers.

3.2.5. Tuning Step

Most of the methods need a tuning step in which internal data structures are created according to the actual system that is simulated. Other methods need information about the distribution of the particles within the system and between the processes. Therefore the tuning step is mandatory. In order to tune the methods, the user has to call the function *fcs_tune* with the function call described in figure (15.3). The function calls awaits several input parameters and is similar to the function call of *fcs_run* described in the following section. First parameter of the tuning is the handle, which was created in *fcs_init* (3.2.3) and then filled with *fcs_common_set* (3.2.4). The parameter *n_locp* gives the number of particles on the local process, which is identical to the length of the array given as parameter *charges*. For the array containing the charges, an array with thrice the length has to be supplied as parameter *charges*. As final parameter, the parameter *n_maxlocp* gives an estimation of the maximum number of particles being stored on the local process before the next call of the tuning routine.

This routine is mandatory to be called, in order to grant a unified interface for the library. Some methods, e.g. PEPC, do not need this tuning step. But to be able to switch the methods within a code without the need to greatly modify the code again, the tuning routine has to be called. If the user wants to use e.g. PEPC as well as e.g. P^{2}NFFT, he can use the same program (except for the method-specific setter routines), when including the tuning routine.

3.2.6. Calculation Step

The calculation step is the centerpiece of the library. Within it, the actual calculation of Coulomb interactions takes place. As mentioned in the description of tuning step, the call to *fcs_run* (fig. 15.4) is nearly the same, as for *fcs_tune* (see 3.2.5). There are two additional parameters, which are *field* and *potential*. The first is the array to which the field data is added and has a size of thrice the number of local particles (*n_locp*), the latter is the array in which the corresponding potentials are saved and has a size equal to the number of local particles (*n_locp*). All the other parameters are equal to the parameters of *fcs_tune* (see above).

3.2.7. Clean Up Step

After the calculations are done, the memory allocated by the methods needs to be freed again. This task is done by the function *fcs_destroy* (fig. 15.5). As parameters only the FCS object is needed, in order to free memory allocated in connection with it, since the internal data structures of the methods are administrated by those. The call to fcs_destroy is not mandatory but the user is advised to call it, especially if he or she wants to repeat the calculation with another method without restarting the program, since it cannot be guaranteed that more then one method can be used simultaneously without memory problems.

3.3. Error Handling

Within the ScaFaCoS library a FCSResult type is defined. With this type return values and error messages of the methods and the interface routines are handled. The type includes up to three pieces of information about error that have occurred during a call to a ScaFaCoS function. These pieces are a return code, an error message and the source of the error within the library. As return values the values given in table 3.5 can occur. In order to simplify the error handling for the ScaFaCoS routines, the routines return *NULL* as return value, instead of an allocated error object containing *FCS_SUCCESS* as return value. The other information contained in the error type gives more information about the kind of error that occurred, e.g. if a method cannot handle a certain system due to restrictions on periodicity, and where the error occurred within the library since the error can happen in functions called inside the routine called by the user.
To get the information out of the error type, there are three functions for this task.

Their calls are shown in figures 15.6 to 15.8. Due to restrictions in Fortran the message length in Fortran is capped to 256 characters.

3.4. Fortran Specifics

```
! Fortran
use iso_c_binding
character(kind = c_char, len = 32) :: string = ”test_string”
string = trim(adjustl(string)) // c_null_char
```

Figure 3.4.: Example for string truncation for strings used with ScaFaCoS

In this section some advises are given for Fortran programmers. Since the Fortran interface is a wrapper interface above the C version of the interface trying to stay as true to the C interface as possible, some things have to be observed. First it is advised to use the interoperable Fortran kinds delivered by the library, which are *fcs_integer_kind_isoc* for integer and *fcs_real_kind_isoc*. With the usage of these kinds problems concerning the interoperability of the values is avoided. Secondly strings have to be terminated with C null characters and have to be trimmed to the correct length. An example how this can be achieved is given in figure 3.4. A third difference from Fortran to C is that in the Fortran interface wherever possible the logical type is used instead of an integer value in C. This is true for every manual setter and getter connected to flags. In the parser this exchange was not possible (see 3.5.4).

3.5. Further Functionality

This sections describes some of the functionality the ScaFaCoS library has, which exceeds the pure calculation of Coulomb interactions. It also describes the output functions within the library and the parser for parameters.

3.5.1. Near Field Solver

Should the near field solver be available to the user?

Within the library a near field solver for given near field potentials is included. Some of the methods (P^2NFFT and P^3M) need to calculate separate near field interactions, which is done by this solver. The solver uses a linked cell scheme for fast interaction calculation and uses the sorting library to create and duplicate the particles (ghost-particles). It is able to handle periodicity and linked cell sizes that are smaller than the cut-off radius. With the near field flag described in section 3.2.4 the calculation of the near field portions of the Coulomb interactions can be delegated to external routines. In order to do this the methods which support this (currently P^3M and P^2NFFT) provide a functions which can then be called from the user's program to calculate the corresponding near field portions of the Coulomb interactions (see figure 15.9 to 15.11). The aforementioned functions are generic function that will calculate the according values for the chosen method. It is possible to check if the chosen method is able to relegate the near field calculation by use of *fcs_method_has_near* (15.12).

3.5.2. Optional Results

It is possible to get additional results from the solver concerning the calculation. As of now the only additional result to have is the system virial. It is possible that some solvers add additional optional results in the future. Since now only the virial is available here will be explained how to get the virial from the solvers. For that two functions are needed, *fcs_require_virial* (15.15) and *fcs_get_virial* (15.16). With *fcs_require_virial* the user activates or deactivates the calculation of the virial. If set to true, the virial is activated, else it is not calculated. If it is calculated, the user can get the calculated by the use of *fcs_get_virial*. For that he has to pass an allocated array of nine *fcs_float* values to the function.
Possibly in the next future other optional results will be implemented, if required by the community.

3.5.3. Output of Interface Parameters

For debugging purposes it is possible to print out the content of the FCS object. This is done by the use of the function *fcs_printContent* (figure 15.13). The output shows the current values of the parameters set within the FCS object. If no changes were made to the solver-specific parameters, it shows the default values for them. The behavior of the routine if used on a freshly created FCS object without set common parameters is undefined.

3.5.4. Parser

To simplify the parameter input for programs basing on script file input, the ScaFaCoS library offers a parser that allows to read in parameters from a string. The format of the passed string must adhere to the example shown in figure 3.5, while the function call can be seen in figure 15.14

```
string = "box_a,1.0,0.0,0.0,near_field_flag,1,pepc_theta,0.3"
```

Figure 3.5.: Example of a parser string setting the first box vector, the near field flag and a method-specific parameter.

The format is an alteration of parameter name and value(s). In the example the first parameter to be set is a generic one, so no prefix is needed and the parameter name is **box_a**. Since the first box vector needs three floating point values, these are given after the parameter name, always separated by commas. After the final parameter value, the next parameter name is expected, in the case of the example the parameter is another generic one, the **near_field_flag**. Because this parameter needs only one integer value, it given and immediately followed by the next parameter name. The last parameter to be set is a solver-specific one, the parameter **theta** from the PEPC solver. Since it is a PEPC-specific parameter, the solver name is added as a prefix, followed by the parameter name. After it the parameter value follows, like with the generic parameters. For Fortran users the advise, that every flag which is set with the parser, has to used the C syntax which requires an integer value (0 for false, 1 for true) instead of *.true.* or *.false.*.

4. FMM – Fast Multipole Method

```
Parameters

  - absrel: absolute or relative energy errors
      0 = relative energy error 10^{-3} (default)
      1 = absolute energy error deltaE
      2 = relative energy error deltaE

  - deltaE: relative or absolute energy error, depending on absrel
      absrel = 1: absolute total energy error
      absrel = 2: relative energy error, 10^{-1}...10^{-14}

  - dipole_correction: type of dipole correction to use (explain!)
     currently ignored (not handed over by interface)
     -1 = no dipole correction
      0 = standard dipole correction
      1 = dipole correction activated
      default value?

Box Shape / Periodicity

  - open boundaries:              full support for any box shape

  - periodic or mixed boundaries: support limited to cubic boxes;
                                  however, for mixed boundaries,
                                  edges in non-periodic directions
                                  may be shorter than periodic edges

Can Delegate Nearfield to MD:  no

Bugs / Missing Features

  - only backup version of virial tensor with periodic boundaries
  - can we lift the requirements for periodic or mixed boundaries
    to approximately cubic boxes, like for PEPC?
  - does not compile with Intel compiler
```

5. MEMD – Maxwell Equation Molecular Dynamics

MEMD is a young method for the calculation of electrostatics. Rather than directly solving the Poisson equation, it performs a discrete quasi electrodynamics simulation on a lattice. It comes with some benefits and some restrictions, which will be described here and should be considered before using the algorithm.

5.1. Description of the method

In most algorithms, the electrostatic problem is formulated as the solution to Poisson's equation $\nabla^2\Phi = -4\pi\rho$. This elliptic partial differential equation requires a global solution in space, which is where the high computational cost of electrostatics arises. It can however be seen as nothing but the static limit of full electrodynamics where the speed of light approaches infinity, and the magnetic fields within the system have completely vanished. If we therefore consider the Gauss law $\nabla\mathbf{E} = 4\pi\rho$ of the Maxwell equations, we arrive at a solution of Poisson's equation while only left with a set of hyperbolic differential equations. The solution of these requires only local operations, no global solution in space.

5.1.1. Equations of motion

Of course, full electrodynamics within a simulation are similarly costly, since the speed of field propagations is several orders of magnitude higher than the typical speed of charges within the system, requesting an unfeasably fine time discretization. However, Maggs and Pasichnyk [9, 10] have shown that, in a Car-Parrinello manner, these degrees of freedom can be brought closer by drastically reducing the wave propagation speed. In the thermodynamic limit, the Coulomb interaction is fully recovered independently of this propagation speed.

Denoting the particle masses with m_i, their charges with q_i, their coordinates and momentum with $\mathbf{r}_i$ and $\mathbf{p}_i$ respectively, the equations of motion for the coupled system of charges and fields read

$$m_i\ddot{\mathbf{r}}_i = -\frac{\partial U}{\partial \mathbf{r}_i} - q_i\mathbf{E} + q_i\mathbf{v}_i \times \mathbf{B} \tag{5.1}$$

$$\mathbf{B} = \frac{1}{c^2}\dot{\mathbf{\Theta}} \tag{5.2}$$

$$\dot{\mathbf{D}} = c^2\,\nabla \times (\nabla \times \mathbf{B}) - \frac{\mathbf{j}}{\varepsilon} \tag{5.3}$$

$$\dot{\mathbf{B}} = -\nabla \times \mathbf{D} \tag{5.4}$$

where $\epsilon(\mathbf{r})$ is the local dielectric constant, c the wave propagation speed (speed of light), $\mathbf{B}$ the magnetic field, $\mathbf{D} = \varepsilon\mathbf{E}$ the electric field, $\mathbf{\Theta}$ an additional degree of freedom for the electric field (where $\nabla \times \mathbf{\Theta} = 0$), and $\mathbf{j}$ the electric current density.

5.1.2. Discretization in space

In this implementation, the given equations are dicretized in space on a regular lattice for numerical evaluation. For the electric currents (and therefore the fields as well), a linear interpolations scheme is introduced because it allows for arbitrary local changes of the dielectric background constant $\varepsilon(\mathbf{R})$. The gradient and curl operators are implemented via finite differences. This linear interpolation leads to a larger numerical error, especially on short ranges, as will be further explored in section 5.1.6.

5.1.3. Discretization in time

The MEMD algorithm is based on moving charges and wave propagation, so unlike all other method in the ScaFaCoS library, it depends on the dynamics of the system. Most notably on the time step of the Molecular Dynamics (MD) integrator and the resulting particle speeds.

The calculation of the electric current and the propagation of the magnetic field are discretized in time. Thus, a time step is required by the user and it should match the integration time step of the simulation. If a different (non-MD) simulation includes the ScaFaCoS library, the MEMD solver can be a problematic choice and should only be considered if you know what you are doing.

To maintain the energy conservation feature of symplectic integrators, the field calculation within this algorithm features a time reversible scheme: First, the magnetic fields are propagated for half a time step, then the electric fields are calculated, and finally the magnetic fields are propagated another half time step.

5.1.4. Boundary conditions

A local algorithm offers the possibility for a real periodic three dimensional torus geometry by simply stitching together the wave propagation of opposite box surfaces. This however implies a real periodic geometry and does not have the possibility to manually neglect the dipole term of the system, like *e.g.* Ewald-based methods do by omitting the dipole term in Fourier space. Even worse, the dipole term of the unfolded coordinate system is considered, since the history of the particle trajectories is stored in the magnetic fields.

This implementation corrects for the unwanted dipole term by directly substracting it, and therefore features metallic boundary conditions at infinity. Multipole terms of higher order decay faster than $1/r^3$ and are of short ranged nature.

5.1.5. Calculation of the potential

The MEMD algorithm does not solve the Poisson equation but only updates the electric field directly (see section 5.1.1). The electrostatic potential is never actually calculated. If output of the potential is wanted, the MEMD algorithm will integrate over the field strength at all particle positions

$$\Phi_{\text{total}} = \frac{1}{2}\int_V \mathbf{E}\cdot\mathbf{D}dV \tag{5.5}$$

The potential energy within the magnetic fields are omitted, since the fields in this algorithm are artificial and only their curl will contribute. This results in variations in the energy since the magnetic fields act as an energy depot in the system, although the total energy is conserved over time.

5.1.6. Error estimate

The main error of the MEMD algorithm stems from the lattice interpolation and is proportional to the lattice size in three dimensions, which means $\Delta_{\text{lattice}} \propto a^3$.

Without derivation here, the algorithmic error is proportional to $1/c^2$, where c is the adjustable wave propagation speed. From the stability criterion, this yields

$$\Delta_{\text{MEMD}} = A \cdot a^3 + B \cdot dt^2/a^2 \tag{5.6}$$

This means that increasing the lattice will help the algorithmic error, as we can set the wave propagation speed to a higher value. At the same time, it increases the interpolation error at an even higher rate. Therefore, momentarily it is advisable to choose the lattice with a rather fine mesh of approximately the size of the particles. The ideal wave propagation can then be tuned by the method.

5.2. Systems suited for the algorithm

As stated in section 5.1.3, MEMD is a dynamic algorithm. It relies heavily on the system moving slowly between calculations and requires a time step as parameter. It should only be used to couple to an MD simulation with a symplectic integrator. With anything else, you might run into unexpected difficulties. It also has to be taken into account that for energy calculations, the results can vary significantly because of the energy stored temporarily in the magnetic fields (see 5.1.5). The algorithm being dynamic also implies that there should be no permanently fixed charges within the system. MEMD works with electric currents, and a charge that does not move can produce unpredictable errors if its magnetic field radiation relaxes to zero over time. If you must feature fixed particles, try to keep them in a narrow potential minimum.

It is also mentioned in section 5.1.4 that MEMD directly substracts the system's dipole moment to deploy metallic boundary conditions at infinite distance. This works fine unless the system is externally driven to create a particle flow (net electric current) in one direction. The dipole moment will then diverge and the system needs to be reinitialized regularly.

Generally, the simulated system

- should be propagated in an MD like manner.
- should be periodic in all dimensions.
- should be of cubic geometry.
- should not be very inhomogenous.
- should not have a net electric current.
- should not contain fixed charges.
- should run for some time to make up for the slow initialization procedure.

These conditions seem very restrictive, but they are given for most straight forward MD simulations. And if given, the algorithm can compete with the other electrostatics methods in this software package.

5.3. Solver-specific parameters

The tolerance parameters are common to all ScaFaCoS solvers. At the moment MEMD only supports automatic tuning to reach the lowest possible error. This is typically in the range of 10^{-3} for the RMS force error.

In addition, MEMD has the following, partly mandatory, parameters, which can be adjusted by the MEMD-specific functions described in the next section. Alternatively, you can use the ScaFaCoS test program together with the optional command line argument `-c` and a comma separated list of parameter settings, *e.g.*, use

```
./scafacos_test memd systems/3d-periodic/cloud_wall.xml.gz \
-c tolerance_field,1e-3,mesh,16,timestep,0.01,lightspeed,0.05
```

- `timestep` (mandatory) – The time step (in simulation units) of the Molecular Dynamics (MD) integrator. This is a mandatory parameter, since without it, MEMD does not have a frame of reference for particle and fiel propagation speeds.

- `mesh` – The space discretization mesh size in one dimension. If not set, this is automatically tuned via the minimal particle pair distance of the first system state.

- `lightspeed` – The propagation speed of the magnetic fields in the system. This parameter is connected via a stability criterion to `timestep` and `mesh`, as explained below this list.

- `permittivity` – The background permittivity ε of the system. Only two of the parameters `permittivity`, `temperature`, and `bjerrum_length` can be set at the same time, since they are mathematically dependent.

- `temperature` – The temperature of the system. This is important for the thermalization of the fields. Only two of the parameters `permittivity`, `temperature`, and `bjerrum_length` can be set at the same time, since they are mathematically dependent.

- `bjerrum_length` – The bjerrum length l_B of the system's dielectric background. Only two of the parameters `permittivity`, `temperature`, and `bjerrum_length` can be set at the same time, since they are mathematically dependent.

The three parameters `timestep`, `mesh` and `lightspeed` are connected via the stability criterion for the algorithm. The condition that the propagation of the magnetic fields has to be significantly larger than the speed of the particles leads to the relation

$$c \ll \frac{a}{dt} \tag{5.7}$$

where c is the propagation speed of the fields (light speed), a is the lattice spacing, and dt is the time step. Usually, the time step is a fixed property. To achieve a good performance with the algorithm, a rule of thumb would be to set the lattice spacing at about the size of the particles (minimal distance), divide the box length by that, and pick the closest power of two for the mesh size in the system. From the resulting lattice spacing and the fixed time step, calculate the right hand fraction in equation (5.7) and multiply it by 0.01 to get a lightspeed estimate for an appropriately stable algorithm with minimal numerical error.

5.4. Solver-specific functions

-
```
FCSResult fcs_memd_set_box_size(FCS handle,
    fcs_float length_x, fcs_float length_y, fcs_float length_z);
FCSResult fcs_memd_set_total_number_of_particles(FCS handle,
    fcs_int number_of_particles);
FCSResult fcs_memd_set_local_number_of_particles(FCS handle,
    fcs_int number_of_particles);
```

  Set system parameters, common for all methods.

- `FCSResult fcs_memd_set_permittivity(FCS handle,`
  `    fcs_float epsilon);`
  `FCSResult fcs_memd_set_temperature(FCS handle,`
  `    fcs_float temperature);`
  `FCSResult fcs_memd_set_bjerrum_length(FCS handle,`
  `    fcs_float bjerrum);`

  Set the dielectric background parameters for the method. See section 5.3 for instructions.

- `FCSResult fcs_memd_set_time_step(FCS handle,`
  `    fcs_float timestep);`
  `FCSResult fcs_memd_set_mesh_size_1D(FCS handle,`
  `    fcs_int mesh_size);`
  `FCSResult fcs_memd_set_speed_of_light(FCS handle,`
  `    fcs_float lightspeed);`

  Set the tuning parameters for the method. See section 5.3 for instructions.

- `FCSResult fcs_memd_set_init_flag(FCS handle,`
  `    fcs_int flagvalue);`

  If the system changes major parameters, the initial solution routine will be called automatically. But if you add small particle numbers or change particle positions rapidly, this function will tell MEMD to re-initialize the system and can prevent a crash.

5.5. Known bugs or missing features

Since the algorithm's main strength is its locality and with it the possibility to apply spatially varying dielectric background properties, the following missing features have not been included in the ScaFaCoS library.

- The initial solution can theoretically be calculated with one of the alternative methods. This would speed up the algorithm significantly. It has not been implemented yet since one would need to distinguish between a constant and varying dielectric background.
- For a constant dielectric background, an option could also be added to interpolate the charges over a wider area and calculate the short-range part of the electrostatic interaction seperately. This would increase the precision but would again not be compatible with varying dielectric properties.

6. MMM1D

7. MMM2D

8. Ewald

The well-known Ewald formula for the computation of (**??**) splits the electrostatic potential ϕ into the following parts

$$\phi = \phi^{\text{real}} + \phi^{\text{reci}} + \phi^{\text{self}} + \phi^{\text{dipo}}, \tag{8.1}$$

where the contribution from real space ϕ^{real}, reciprocal space ϕ^{reci}, the self-energy ϕ^{self} and the dipole correction ϕ^{dipo} are given by

$$\begin{aligned}
\phi^{\text{real}}(\mathbf{x}_j) &= \sum_{\mathbf{r}\in\mathbb{Z}^3} \sum_{\substack{l=1\\ l\neq j \text{ for } \mathbf{r}=\mathbf{0}}}^{M} q_l \frac{\operatorname{erfc}(\alpha\|\mathbf{x}_j-\mathbf{x}_l+\mathbf{r}B\|_2)}{\|\mathbf{x}_j-\mathbf{x}_l+\mathbf{r}B\|_2}, \\
\phi^{\text{reci}}(\mathbf{x}_j) &= \frac{1}{\pi B} \sum_{\mathbf{k}\in\mathbb{Z}^3\setminus\{\mathbf{0}\}} \frac{\mathrm{e}^{-\pi^2\|\mathbf{k}\|_2^2/(\alpha B)^2}}{\|\mathbf{k}\|_2^2} \sum_{l=1}^{M} q_l \mathrm{e}^{-2\pi\mathrm{i}\mathbf{k}(\mathbf{x}_j-\mathbf{x}_l)/B}, \\
\phi^{\text{self}}(\mathbf{x}_j) &= -2q_j\frac{\alpha}{\sqrt{\pi}}, \\
\widetilde{\phi}^{\text{dipo}} &= \frac{2\pi}{3V}\left(\sum_{l=1}^{M} q_l\mathbf{x}_l\right)^2.
\end{aligned} \tag{8.2}$$

Thereby, the complementary error function is defined by $\operatorname{erfc}(z) = \frac{2}{\sqrt{\pi}}\int_z^\infty \mathrm{e}^{-t^2}\mathrm{d}t$. Choosing optimal parameters, Ewald summation scales as $\mathcal{O}(M^{3/2})$ [8].

9. P^{2}NFFT– Particle-Particle NFFT

P^{2}NFFT is a common framework for almost all FFT based fast Coulomb solvers. The computation of Coulomb interactions is split into a short range interaction (near field) and a long range interaction (far field). For sake of simplicity, we describe the idea of P^{2}NFFT only for the one-dimensional case. In addition, we do not stress all the details, that arise from the difference between periodic and non-periodic boundary conditions. More detailed descriptions of the algorithms can be found in [11].

9.1. Description of the Method

As usual, we assume M charges q_l at position r_l. The potentials ϕ_j at position r_j are given by

$$\phi_j = {\sum_{l=1}^{M}}' q_l \frac{1}{r_{jl}} \,,$$

and the Electrostatic fields E_j at position r_j are given by

$$E_j = {\sum_{l=1}^{M}}' q_l \nabla \frac{1}{r_{jl}} = {\sum_{l=1}^{M}}' q_l \frac{r_l}{r_{jl}} \,.$$

Hereby, the distance between two particles at positions r_j and r_l is given by $r_{jl} := |r_j - r_l|$.

9.1.1. Periodic boundary conditions

In this section we present a straightforward method, that accelerates the traditional Ewald summation technique by NFFT. This combination was first presented in [7] and is very similar to the FFT-accelerated Ewald sums, namely, the so-called particle-particle particle-mesh (P^{3}M), particle-mesh Ewald (PME) and smooth particle-mesh Ewald (SPME), see also [5]. Additionally we will see, that the accelerated Ewald summation can be reinterpreted into a method very similar to our fastsum Algorithm **??**.

In order to overcome this increase in time we apply the NFFT for the calculation of the reciprocal-space potential ϕ^{reci} and we obtain a method similar as our fast summation method. To this end, we compute the Fourier transformed charge density

$$S(\mathbf{k}) = \sum_{l=1}^{M} q_l \mathrm{e}^{+2\pi \mathrm{i} \mathbf{k} \mathbf{x}_l / B}$$

by NFFT$^{\mathsf{H}}$ and after truncation of the sum (8.2) we obtain by NFFT

$$\phi^{\mathrm{reci}}(\mathbf{x}_j) \approx \frac{1}{\pi B} \sum_{\mathbf{k}\in I_{\mathbf{N}}\setminus\{\mathbf{0}\}} \frac{\mathrm{e}^{-\pi^2\|\mathbf{k}\|_2^2/(\alpha B)^2}}{\|\mathbf{k}\|_2^2} S(\mathbf{k}) \mathrm{e}^{-2\pi\mathrm{i}\mathbf{k}\mathbf{x}_j/B} .$$

9.1.2. Calculation of the Potentials

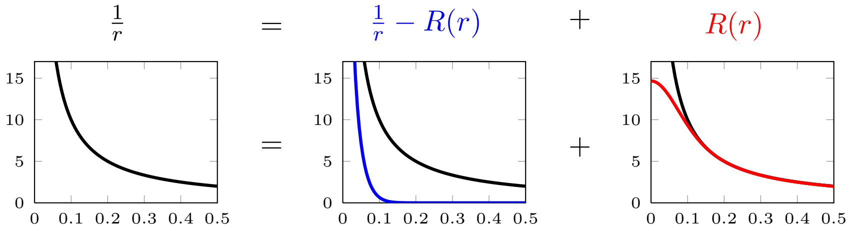

Figure 9.1.: Splitting of Coulomb potential into near field (blue) and far field (red).

$$\phi(r_j) = \sum_{l=1}^{M}{}' \frac{q_l}{r_{jl}} = R(0) + \sum_{l=1}^{M}{}' q_l \left(\frac{1}{r_{jl}} - R(r_{jl}) \right) - \sum_{l=1}^{M} q_l R(r_{jl}) , \quad j = 1, \dots, M$$

$$R(r) \approx \sum_{k=-N/2}^{N/2-1} \hat{R}_k \mathrm{e}^{-2\pi\mathrm{i}kr}$$

$$\sum_{l=1}^{M} q_l R(r_{jl}) \approx \sum_{l=1}^{M} q_l \sum_{k=-N/2}^{N/2-1} \hat{R}_k \mathrm{e}^{-2\pi\mathrm{i}kr_{jl}} = \sum_{k=-N/2}^{N/2-1} \hat{R}_k \left(\sum_{l=1}^{M} q_l \mathrm{e}^{+2\pi\mathrm{i}kr_l} \right) \mathrm{e}^{-2\pi\mathrm{i}kr_j}$$

Near field interactions are computed with the ScaFaCoS near field solver, but can also be redirected to any other the near field solver.

Far field interactions are computed via convolution in Fourier space. With the appropriate choice of window functions and convolutions in Fourier space, P^{2}NFFT becomes the P3M (default for fully periodic boundaries), the NFFT based fast summation (default for fully non-periodic boundaries) or almost any other FFT based fast Coulomb solver like PME, SPME, Gaussian split Ewald or plain Ewald summation. Note, that an uniform particle distribution is required in order to achieve the typical $\mathcal{O}(N \log N)$ scaling.

9.1.3. Calculation of the Electrostatic Fields

9.1.4. Calculation of the Virial

9.2. Features

Periodicity: Only fully periodic and fully non-periodic boundaries are supported.

Box shape: Only cubic box shape is supported.

Autotuning: For fully periodic boundaries, the parameters can be automatically tuned when a required tolerance in the absolute rms field error is provided. For fully non-periodic boundaries, the parameters are more or less intelligently guessed when a required tolerance in the absolute potential error is provided.

Delegate near-field: Yes.

Virial: Yes for fully non-periodic boundaries. For fully periodic boundaries we only compute an approximation of the diagonal of the virial tensor, *i.e.*, all entries of the diagonal are set equal to one third of the total energy.

9.3. Solver-specific Parameters

The tolerance parameters are common to all ScaFaCoS solvers. At the moment P^2NFFT only supports automatic parameter tuning for the following tolerance types.

- `tolerance_type` - The type of error that we want to check for. Allowed values are `FCS_TOLERANCE_TYPE_FIELD` (absolute rms error in the fields) for fully periodic boundaries and `FCS_TOLERANCE_TYPE_POTENTIAL` (absolute rms error in the potentials) for non-periodic boundaries.
- `tolerance` - The allowed tolerance of the error. Type of the error is given by `tolerance_type`. If the required tolerance can't be achieved with the given parameters, P^2NFFT aborts with an error message.

In addition, P^2NFFT offers the following parameters, which can be adjusted by the P^2NFFT-specific functions described in the next section. Alternatively, you can use the ScaFaCoS test program together with the optional command line argument `-c` and a comma separated list of parameter settings, e.g., use

```
./scafacos_test p2nfft systems/3d-periodic/cloud_wall.xml.gz \
-c tolerance_field,1e-4,p2nfft_r_cut,4.5,pnfft_N,16,16,16,pfft_patience,0
```

in order to run the test program with a near field cut off range of 4.5 and a sloppy planned FFT of size $16 \times 16 \times 16$ up to tolerance 10^{-4} in the fields.

- `p2nfft_r_cut` - Absolute cutoff distance of the near field. Can be automatically tuned. Feasible values are any positive floats. Especially, P^2NFFT still works correct if `p2nfft_r_cut` exceeds the simulation box. We do not require that `p2nfft_r_cut` has to be less than half the smallest simulation box length. When `p2nfft_r_cut` is chosen too small, the errors of the algorithm become large, the required accuracy might not be achieved, or the grid has to be chosen very large, which will slow down the algorithm. When `p2nfft_r_cut` is chosen too large, the algorithm might become slow as most computation is done in the near field region. This parameter corresponds to `r_cut` of the P3M solver.

- `p2nfft_epsI` - Relative cutoff distance of the near field. P^2NFFT scales the original box size into a cube of side length 1. For fully periodic boundary conditions this corresponds to a simple scaling by the box size. However, the scaling is more complicated if non-periodic boundary conditions are involved. Application of the same scaling factor on `r_cut` gives `epsI`. Feasible values are any positive floats that fulfill `epsI` < 0.5.
- `p2nfft_grid` (acronym: `pnfft_N`) - Size of the FFT grid. This is a acronym for `pnfft_N` and is used in the same way. For details look at the PNFFT-specific parameters list. This parameter corresponds to the FFT grid size (`p3m_grid`) of the P3M solver.
- `p2nfft_oversampled_grid` (acronym: `pnfft_n`) - Size of the oversampled FFT grid. This is a acronym for `pnfft_n` and is used in the same way. For details look at the PNFFT-specific parameters list.
- `p2nfft_alpha` - Ewald splitting parameter α. Can be automatically tuned. This parameter balances the errors of the Ewald splitting corresponding to the real space and Fourier space contributions. Choosing *alpha* too small (too large) means that the real space (Fourier space) error becomes dominating. This parameter corresponds to `alpha` of the P3M solver.
- `p2nfft_cao` (see also: `pnfft_m`)- Charge assignment order. Can be automatically tuned. This parameter mostly corresponds to `cao` of the P3M solver. However, note that this is only a wrapper that sets `pnfft_m` to the value `(p2nfft_cao+1)/2` (division rounded down). Therefore, P^2NFFT will always use an even charge assignment order.
- `p2nfft_reg_near` - Choose between the two near field regularization methods for non-periodic boundaries. Feasible values are 0 (Fourier coefficients have been precomputed via CG iteration, only available for some sets of parameters) and 1 (Regularization with two-point Taylor polynomials). Default value is 0, if possible.
- `p2nfft_reg_near_name` (alternative: `p2nfft_reg_near`) - Choose between the two regularization methods for non-periodic boundaries. Feasible values are `cg` (Fourier coefficients have been precomputed via CG iteration, only available for some sets of parameters) and `t2p` (Regularization with two-point Taylor polynomials). Default is `cg`, if possible.
- `p2nfft_reg_far` - Choose between the available regularization methods at the far field boundary for non-periodic boundaries. Feasible values are 0 (Fourier coefficients have been precomputed via CG iteration, only available for some sets of

parameters and only available in combination with the flag `rad_cg` for near field regularization), 1 (radial regularization with symmetric two-point Taylor polynomials), 2 (radial regularization with two-point Taylor polynomials and explicitly chosen constant continuation value; see `p2nfft_c`), 3 (radial regularization with two-point Taylor polynomials and implicitly chosen constant continuation value), 4 (rectangular regularization with symmetric two-point Taylor polynomials), 5 (rectangular regularization with two-point Taylor polynomials and explicitly chosen constant continuation value; see `p2nfft_c`), and 6 (rectangular regularization with two-point Taylor polynomials and implicitly chosen constant continuation value). Default value is 3.

- `p2nfft_reg_far_name` (alternative: `p2nfft_reg_far`) - Choose between the two regularization methods for non-periodic boundaries. Feasible values are `rad_cg` (radial regularization based on the `cg` near field regularization, only available for some sets of parameters), `rad_t2p_sym` (radial regularization with symmetric two-point Taylor polynomial), `rad_t2p_ec` (radial regularization with two-point Taylor polynomial and explicitly chosen constant continuation value; see `p2nfft_c`), `rad_t2p_ic` (radial regularization with two-point Taylor polynomial and implicitly chosen constant continuation value), `rec_t2p_sym` (rectangular regularization with symmetric two-point Taylor polynomial), `rec_t2p_ec` (rectangular regularization with two-point Taylor polynomial and explicitly chosen constant continuation value; see `p2nfft_c`), and `rec_t2p_ic` (rectangular regularization with two-point Taylor polynomial and implicitly chosen constant continuation value). Default value is `rad_t2p_ic`.
- `p2nfft_epsB` - Regularization border for P^2NFFT with non-periodic boundary conditions. This parameter corresponds to the rescaled system into a unit cube of side length 1. Feasible values are any positive floats that fulfill $\texttt{epsI} + \texttt{epsB} < 0.5$. This parameter does not have any effect for fully periodic boundary conditions. Default value is $\texttt{epsB} := \texttt{epsI}$.
- `p2nfft_p` - The degree of the two-point Taylor polynomial. This parameter only takes affect, if regularization `t2p` has been enabled. Feasible values are any integers between 1 and 16. Default value is 7.
- `p2nfft_c` - The constant continuation value of the far field regularization. This parameter only takes affect, if far field regularization `rad_t2p_ec` or `rec_t2p_ec` is enabled. Feasible values are any floating point numbers. Default value is 0.0.
- `p2nfft_require_virial` - Decide if the computation of the virial should be included in P^2NFFT. Feasible values are 0 (disable virial computation), or any other integer (enable virial computation). Default value is 0 (disable virial computation).
- `p2nfft_intpol_order` - P^2NFFT uses interpolation tables to speed up the repeated computation of the near field correction. The order of interpolation is

given by `intpol_order`. Feasible values are -2 (approximation of the error function according to [4, eq. (7.1.26)], which yields an error of 1.5×10^{-7}), -1 (direct evaluation of the error function), 0 (constant interpolation), 1 (linear interpolation), 2 (quadratic interpolation) and 3 (cubic interpolation). Default value is 3.

- `p2nfft_k_cut` - Spherical cutoff distance in Fourier space. If this parameter is set to a positive float $k_c > 0$, all Fourier coefficients $\hat{f}_{\mathbf{k}}$ with $\|\mathbf{k}\|_2 > kc$ will be set to zero, i.e., the approximation in Fourier space uses a spherical cutoff scheme. This is usefull, e.g., for comparision of results to classical Ewald summation. Feasible values are any positive float (enable spherical cutoff) or any negative float (disable spherical cutoff). The spherical cutoff is not applied per default, since FFTs naturally use a box shaped cutoff scheme.

- `p2nfft_ignore_tolerance` - On default, P^{2}NFFT aborts with an error message, if the required error tolerance can not be reached. This flag disables the check for accuracy and gives the possibility to run P^{2}NFFT with any set of parameters. It is intended for very experienced users that tune all parameters manually. Default value is 0 (abort if tolerance check fails). Any other value disables the tolerance check.

- `p2nfft_ignore_field` - Disable the computation of the field during `fcs_run`. Default value is 0 (compute field). Any other integer value disables the field computation.

- `p2nfft_verbose_tuning` - Decide if a full list of P^{2}NFFT parameter settings is printed on stdout or not. Feasible values are 0 (do not print), or any other integer (print the parameters). The default value is 1 if ScaFaCoS was configured with the `--enable-fcs-info` flag, and 0 otherwise.

9.3.1. PNFFT-specific Parameters

- `pnfft_N` (acronym: `p2nfft_grid`) - Size of the FFT grid. Can be automatically tuned. Feasible values are three comma separated, even, positve integers, e.g., `pnfft_N,16,32,12`. Optimal values are in the order of $M^{1/3}$, *i.e.*, one grid point for each particle. The grid size may be any even, positive integer, but powers of two are recommended. The larger the grid, the smaller the error, but also the higher the memory and computational requirements of the algorithm. This parameter corresponds to the FFT grid size (`grid`) of the P3M solver.

- `pnfft_n` (acronym: `p2nfft_oversampled_grid`) - Size of the oversampled FFT grid. Can be automatically tuned. Especially for non-periodic boundaries it is necessary to introduce oversampling to reduce the aliasing error. Feasible values are three comma separated, even integers with `n` $\geq$ `N` in every dimension, e.g., `pnfft_N,16,32,12,pnfft_n,20,64,12`. Default value is `N` for fully periodic boundaries and 2`N` for fully non-periodic boundaries.

- `pnfft_window` (alternative: `pnfft_window_name`) - NFFT window function. Feasible values are 0 (Kaiser-Bessel window), 1 (Gaussian window), 2 (B-spline window), 3 (Sine Cardinal window - Fourier transform of the B-spline window) and 4 (window function based on the modified Bessel function of first kind - Fourier transform of the Kaiser-Bessel window). Default value is 2 (B-spline). Recommended values are 0 (Kaiser-Bessel) for fully non-periodic boundaries and 2 (B-spline) for fully periodic boundaries.

- `pnfft_window_name` (alternative: `pnfft_window`) - Name of the NFFT window function. Feasible values are `kaiser` (Kaiser-Bessel window), `gaussian` (Gaussian window), `bspline` (B-spline window), `sinc` (Sine Cardinal window - Fourier transform of the B-spline window) and `bessel_i0` (window function based on the modified Bessel function of first kind - Fourier transform of the Kaiser-Bessel window). Default value is `bspline`. Recommended values are `kaiser` for fully non-periodic boundaries and `bspline` for fully periodic boundaries.

- `pnfft_m` (see also: `p2nfft_cao`) - Real space cutoff of the window function. The number of grid points that are influenced by a charged particle. Can be automatically tuned. Allowed values are any positive integers. In most cases `m` is chosen between 1 (low precision) and 16 (very high precision).
  Note: Due to historical reasons, this parameter corresponds to one half of the charge assignment order (`cao`) of the P3M solver!

- `pnfft_b` - Shape parameter of the NFFT window function corresponding to the three dimensions. Can be automatically tuned. Feasible values are three comma separated positiv floating point numbers, e.g., `pnfft_b,0.5,1.0,3.2`. Choosing an optimal value for b is not that easy but influences the error a lot. Therefore, you should only change the default value if you know what you do.

- `pnfft_intpol_order` - PNFFT uses interpolation tables to speed up the repeated evaluation of the window functions. The integer `pnfft_intpol_order` gives the interpolation order. Feasible values are -1 (direct evaluation of window function without interpolation), 0, (constant interpolation), 1 (linear interpolation), 2 (quadratic interpolation) and 3 (cubic interpolation). Default value is 3 (cubic interpolation).

- `pnfft_direct` - Flag to enable plain NDFT instead of the fast approximative NFFT algorithms. Feasible values are 0 (use NFFT), or any other integer `pnfft_direct`$\neq 0$ (use NDFT). Default value is `pnfft_direct`$= 0$ (use NFFT).

- `pnfft_pre_phi_hat` - Flag to enable precomputed Fourier coefficients for faster computation of the diagonal matrix in the NFFT algorithm. Feasible values are 0 (turn off pre-computation), or any other integer `pnfft_pre_phi_hat`$\neq 0$ (enable pre-computation). Default value is 1 (enable pre-computation).

- `pnfft_fg_psi` - Set PNFFT flag `PNFFT_FG_PSI`. Only Gaussian window function we can use Fast Gaussian Gridding in order to speed up the evaluation of the window function. Feasible values are 0 (switch Fast Gaussian gridding off), or any other integer `pnfft_fg_psi`$\neq 0$ (use Gaussian gridding). Default value is 0 (since B-Spline is the default window function).

- `pnfft_fft_in_place` - Set PNFFT flag `PNFFT_FFT_IN_PLACE`. This causes the PNFFT planner to use in-place-FFTs, which saves about half of the memory for FFT but may sacrifices some performance. Feasible values are 0 (call out-of-place FFTs), or any other integer `pnfft_fft_in_place`$\neq 0$ (call in-place-FFTs). Default value is `pnfft_fft_in_place`$= 0$ (call out-of-place FFTs).

- `pnfft_sort_nodes` - Set PNFFT flag `PNFFT_SORT_NODES`. Chooses whether to call a local sort before the interpolation step. This may result in better cache locality. Feasible values are 0 (omit local sort), or any other integer `pnfft_sort_nodes`$\neq 0$ (sort before interpolation). Default value is `pnfft_sort_nodes`$= 0$ (omit local sort).

- `pnfft_interlaced` - Set PNFFT flag `PNFFT_INTERLACED`. Chooses whether PNFFT uses interlacing. This gives better accuracy at the price of more operations. Feasible values are 0 (omit interlacing), or any other integer `pnfft_interlaced`$\neq 0$ (enable interlacing). Default value is `pnfft_interlaced`$= 0$ (omit interlacing).

- `pnfft_grad_ik` - Set PNFFT flag `PNFFT_GRAD_IK`. Chooses whether PNFFT computes the gradient through multiplication with $-2\pi i\mathbf{k}$ in Fourier space. This gives better accuracy in comparison to analytic differentiation at the price of more operations. Especially, we have to call four backward FFTs (instead of one for analytic differentiation). Feasible values are 0 (use analytic differentiation), or any other integer `pnfft_grad_ik`$\neq 0$ (use Fourier space differentiation). Default value is `pnfft_grad_ik`$= 0$ (use analytic differentiation).

- `pnfft_pre_psi` - Set PNFFT flag `PNFFT_PRE_PSI`. Tensor product based precomputation of the window function. This option reduces the amount of repeated evaluations of the window function at the cost of $6(2m+1)M$ floats (or $3(2m+1)M$ floats if the gradient is not needed). Since the this kind of precomputation depends on the nodes $\mathbf{x}_j$, it must be performed during `fcs_run` instead of `fcs_tune`. Feasible values are 0 (switch off precomputation), or any other integer `pnfft_pre_psi`$\neq 0$ (switch on precomputation). Default value is 0 (no precomputation). This flag can not be used together with `pnfft_pre_full_psi`.

- `pnfft_pre_fg_psi` - Set PNFFT flags `PNFFT_FG_PSI` and `PNFFT_PRE_PSI`, i.e., use Fast Gaussian Gridding for tensor product based precomputation. Since the this kind of precomputation depends on the nodes $\mathbf{x}_j$, it must be performed during `fcs_run` instead of `fcs_tune`. Feasible values are 0 (switch off precomputation), or any other integer `pnfft_pre_fg_psi`$\neq 0$ (switch on precomputation).

Default value is 0 (no precomputation). This flag can not be used together with `pnfft_pre_full_psi`.

- `pnfft_pre_full_psi` - Set PNFFT flag `PNFFT_PRE_FULL_PSI`. Full precomputation of the window function. This option reduces the amount of repeated evaluations of the window function at the cost of $4(2m+1)^3M$ floats (or $(2m+1)^3M$ floats if the gradient is not needed). Since the this kind of precomputation depends on the nodes $\mathbf{x}_j$, it must be performed during `fcs_run` instead of `fcs_tune`. Feasible values are 0 (switch off precomputation), or any other integer `pnfft_pre_psi`$\neq 0$ (switch on precomputation). Default value is 0 (no precomputation). This flag can not be used together with `pnfft_pre_full_psi`.

- `pnfft_pre_full_fg_psi` - Set PNFFT flags `PNFFT_FULL_FG_PSI` and `PNFFT_PRE_FULL_PSI`, i.e., use Fast Gaussian Gridding for full window function precomputation. Since the this kind of precomputation depends on the nodes $\mathbf{x}_j$, it must be performed during `fcs_run` instead of `fcs_tune`. Feasible values are 0 (switch off precomputation), or any other integer `pnfft_pre_full_fg_psi`$\neq 0$ (switch on precomputation). Default value is 0 (no precomputation). This flag can not be used together with `pnfft_pre_psi`.

- `pnfft_real_f` - Set PNFFT flag `PNFFT_REAL_F`. Normally, PNFFT works on complex valued inputs. This flag enables some optimizations for real valued inputs, e.g., substituting complex additions and multiplications with real ones. However, this comes at the price of strided memory access and does not always improve performance. Feasible values are 0 (use complex operations), or any other integer `pnfft_real_f`$\neq 0$ (use real operations where possible). Default value is `pnfft_real_f`$= 0$ (use complex operations).

9.3.2. PFFT-specific Parameters

- `pfft_patience` - Similar to FFTW, the PFFT library splits the computation of parallel FFT into a more or less time consuming planning step and a fast execution step. The time spent for planning can be adjusted by the `pfft_patience` flag. Feasible values are 0 (plan PFFT with `PFFT_ESTIMATE`), 1 (plan PFFT with `PFFT_MEASURE`), 2 (plan PFFT with `PFFT_PATIENT`) and 3 (plan PFFT with `PFFT_EXHAUSTIVE`). All other values raise an error. Default value is 1 (`PFFT_MEASURE`).

- `pfft_tune` - The PFFT library uses FFTW for computing serial FFTs combined with serial transpositions. Sometimes its better to perform the FFT and transposition in two separate steps, but the FFTW planner does not recognize. For value `pfft_tune`$= 0$ PFFT uses the FFTW plan as it is. For any other value, PFFT calls an additional planner in order to decide if performance gets better when the local FFT and transposition is performed in two separate steps. For large local array sizes this tuning gets very time consuming. Default value is 0 (tuning turned off).

- `pfft_preserve_input` - PFFT can chose between a larger set of algorithms, if it is allowed to overwrite the input array. Feasible values are 0 (PFFT is allowed to destroy input), or any other integer `pfft_preserve_input`$\neq 0$ (PFFT is not allowed to destroy input). Default value is 0 (PFFT is allowed to destroy input).

9.4. Solver-specific Functions

-
  ```
  FCSResult fcs_p2nfft_set_r_cut(FCS handle, fcs_float r_cut);
  FCSResult fcs_p2nfft_get_r_cut(FCS handle, fcs_float* r_cut);
  FCSResult fcs_p2nfft_set_r_cut_tune(FCS handle);
  ```

  Set/retrieve/restore spherical near field cutoff radius.

-
  ```
  FCSResult fcs_p2nfft_set_epsI(FCS handle, fcs_float eps_I);
  FCSResult fcs_p2nfft_get_epsI(FCS handle, fcs_float* eps_I);
  FCSResult fcs_p2nfft_set_epsI_tune(FCS handle);
  ```

  Set/retrieve/restore normalized near field cutoff radius.

-
  ```
  FCSResult fcs_p2nfft_set_grid(
      FCS handle, fcs_int N0, fcs_int N1, fcs_int N2);
  FCSResult fcs_p2nfft_get_grid(
      FCS handle, fcs_int* N0, fcs_int* N1, fcs_int* N2);
  FCSResult fcs_p2nfft_set_grid_tune(FCS handle);
  ```

  Set/retrieve/restore FFT grid size.

-
  ```
  FCSResult fcs_p2nfft_set_oversampled_grid(
      FCS handle, fcs_int n0, fcs_int n1, fcs_int n2);
  FCSResult fcs_p2nfft_get_oversampled_grid(
      FCS handle, fcs_int* n0, fcs_int* n1, fcs_int* n2);
  FCSResult fcs_p2nfft_set_oversampled_grid_tune(FCS handle);
  ```

  Set/retrieve/restore oversampled FFT grid size.

-
  ```
  FCSResult fcs_p2nfft_set_alpha(FCS handle, fcs_float alpha);
  FCSResult fcs_p2nfft_get_alpha(FCS handle, fcs_float* alpha);
  FCSResult fcs_p2nfft_set_alpha_tune(FCS handle);
  ```

  Set/retrieve/restore Ewald splitting parameter.

-
  ```
  FCSResult fcs_p2nfft_set_cao(FCS handle, fcs_int cao);
  FCSResult fcs_p2nfft_get_cao(FCS handle, fcs_int* cao);
  FCSResult fcs_p2nfft_set_cao_tune(FCS handle);
  ```

  Set/retrieve/restore charge assignment order.

-
  ```
  FCSResult fcs_p2nfft_set_reg_near(FCS handle, fcs_int reg);
  FCSResult fcs_p2nfft_set_reg_near_by_name(
  ```

```
    FCS handle, char* reg_name);
FCSResult fcs_p2nfft_get_reg_near(FCS handle, fcs_int* reg);
```

Set by number/set by name/retrieve number of near field regularization (default = 0). Feasible values are 0 (`"cg"`), and 1 (`"t2p"`).

-
  ```
  FCSResult fcs_p2nfft_set_reg_far(FCS handle, fcs_int reg);
  FCSResult fcs_p2nfft_set_reg_far_by_name(
      FCS handle, char* reg_name);
  FCSResult fcs_p2nfft_get_reg_far(FCS handle, fcs_int* reg);
  ```

  Set by number/set by name/retrieve number of near field regularization (default = 0). Feasible values are 0 (`"rad_cg"`), 1 (`"rad_t2p_sym"`), 2 (`"rad_t2p_ec"`), 3 (`"rad_t2p_ic"`), 4 (`"rec_t2p_sym"`), 5 (`"rec_t2p_ec"`), and 6 (`"rec_t2p_ic"`).

-
  ```
  FCSResult fcs_p2nfft_set_epsB(FCS handle, fcs_float eps_B);
  FCSResult fcs_p2nfft_get_epsB(FCS handle, fcs_float* eps_B);
  FCSResult fcs_p2nfft_set_epsB_tune(FCS handle);
  ```

  Set/retrieve/restore normalized far field regularization border.

-
  ```
  FCSResult fcs_p2nfft_set_p(FCS handle, fcs_int p);
  FCSResult fcs_p2nfft_get_p(FCS handle, fcs_int* p);
  FCSResult fcs_p2nfft_set_p_tune(FCS handle);
  ```

  Set/retrieve/restore polynomial degree of two-point Taylor regularization.

-
  ```
  FCSResult fcs_p2nfft_set_c(FCS handle, fcs_float c);
  FCSResult fcs_p2nfft_get_c(FCS handle, fcs_float* c);
  FCSResult fcs_p2nfft_set_c_tune(FCS handle);
  ```

  Set/retrieve/restore explicit continuation value for two-point Taylor regularization.

-
  ```
  FCSResult fcs_p2nfft_require_virial(
      FCS handle, fcs_int require_virial);
  ```

  Enable virial computation (optional, default = 0).

  ```
  FCSResult fcs_p2nfft_get_virial(FCS handle, fcs_float* virial);
  ```

  Retrieve virial (optional, default)

  ```
  FCSResult fcs_p2nfft_virial_is_active(
      FCS handle, fcs_int* yes_or_no);
  ```

  Check if virial computation is enabled.

-
  ```
  FCSResult fcs_p2nfft_set_interpolation_order(
      FCS handle, fcs_int intpol_order);
  FCSResult fcs_p2nfft_get_interpolation_order(
  ```

```
    FCS handle, fcs_int* intpol_order);
```

Set/retrieve interpolation order of near field correction (optional, default = 3).

-
  ```
  FCSResult fcs_p2nfft_set_k_cut(FCS handle, fcs_float k_cut);
  FCSResult fcs_p2nfft_get_k_cut(FCS handle, fcs_float* k_cut);
  FCSResult fcs_p2nfft_set_k_cut_tune(FCS handle);
  ```

  Set/retrieve/restore spherical far field cutoff radius.

-
  ```
  FCSResult fcs_p2nfft_set_ignore_tolerance(
      FCS handle, fcs_int set_ignore_tolerance);
  FCSResult fcs_p2nfft_get_ignore_tolerance(
      FCS handle, fcs_int* set_ignore_tolerance);
  ```

  Set/retrieve flag for disabling the accuracy check (default = 0).

-
  ```
  FCSResult fcs_p2nfft_set_ignore_field(
      FCS handle, fcs_int set_ignore_field);
  FCSResult fcs_p2nfft_get_ignore_field(
      FCS handle, fcs_int* set_ignore_field);
  ```

  Set/retrieve flag for disabling the field computation (default = 0).

-
  ```
  FCSResult fcs_p2nfft_set_verbose_tuning(
      FCS handle, fcs_int set_verbose_tuning);
  FCSResult fcs_p2nfft_get_verbose_tuning(
      FCS handle, fcs_int* set_verbose_tuning);
  ```

  Set/retrieve flag for verbose tuning. The default value will be 1 (library was configured with `--enable-fcs-info`) or 0 (otherwise).

9.4.1. PNFFT-specific Functions

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_N(
      FCS handle, fcs_int N0, fcs_int N1, fcs_int N2);
  FCSResult fcs_p2nfft_get_pnfft_N(
      FCS handle, fcs_int* N0, fcs_int* N1, fcs_int* N2);
  FCSResult fcs_p2nfft_set_pnfft_N_tune(FCS handle);
  ```

  Set/retrieve/restore FFT grid size.

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_n(
      FCS handle, fcs_int n0, fcs_int n1, fcs_int n2);
  FCSResult fcs_p2nfft_get_pnfft_n(
      FCS handle, fcs_int* n0, fcs_int* n1, fcs_int* n2);
  FCSResult fcs_p2nfft_set_pnfft_n_tune(FCS handle);
  ```

  Set/retrieve/restore oversampled FFT grid size.

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_window(
      FCS handle, fcs_int window);
  FCSResult fcs_p2nfft_get_pnfft_window(
      FCS handle, fcs_int* window);
  FCSResult fcs_p2nfft_set_pnfft_window_by_name(
      FCS handle, char* window_name );
  ```

  Set/retrieve `pnfft_window` by number or name. (optional, default = 1). Feasible values are 0 (`"gaussian"`), 1 (`"bspline"`), 2 (`"sinc"`), 3 (`"kaiser"`) and 4 (`"bessel_i0"`).

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_m(FCS handle, fcs_int m);
  FCSResult fcs_p2nfft_get_pnfft_m(FCS handle, fcs_int* m);
  FCSResult fcs_p2nfft_set_pnfft_m_tune(FCS handle);
  ```

  Set/retrieve/restore real space cutoff of window function.

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_b(FCS handle, fcs_float b0, fcs_float b1, fcs_float b2);
  FCSResult fcs_p2nfft_get_pnfft_b(FCS handle, fcs_float* b0, fcs_float* b1, fcs_float* b
  FCSResult fcs_p2nfft_set_pnfft_b_tune(FCS handle);
  ```

  Set/retrieve/restore shape parameters of window function corresponding to each of the three dimensions.

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_interpolation_order(
      FCS handle, fcs_int intpol_order);
  FCSResult fcs_p2nfft_get_pnfft_interpolation_order(
      FCS handle, fcs_int* intpol_order);
  ```

  Set/retrieve `pnfft_intpol_order` (optional, default = 3).

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_direct(
      FCS handle, fcs_int yes_or_no);
  FCSResult fcs_p2nfft_get_pnfft_direct(
      FCS handle, fcs_int* yes_or_no);
  ```

  Set/retrieve flag that determines if plain NDFT is used instead of NFFT. (optional, default = 0).

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_pre_phi_hat(
      FCS handle, fcs_int yes_or_no);
  FCSResult fcs_p2nfft_get_pnfft_pre_phi_hat(
      FCS handle, fcs_int* yes_or_no);
  ```

  Set/retrieve flag `pnfft_pre_phi_hat` (optional, default = 0).

-
  ```
  FCSResult fcs_p2nfft_set_pnfft_fg_psi(
      FCS handle, fcs_int yes_or_no);
  FCSResult fcs_p2nfft_get_pnfft_fg_psi(
      FCS handle, fcs_int* yes_or_no);
  ```

Set/retrieve flag `pnfft_fg_psi` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_fft_in_place(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_fft_in_place(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_fft_in_place` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_sort_nodes(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_sort_nodes(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_sort_nodes` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_interlaced(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_interlaced(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_interlaced` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_grad_ik(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_grad_ik(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_grad_ik` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_pre_psi(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_pre_psi(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_pre_psi` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_pre_fg_psi(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_pre_fg_psi(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_pre_fg_psi` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_pre_full_psi(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_pre_full_psi(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_pre_full_psi` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_pre_full_fg_psi(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_pre_full_fg_psi(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_pre_full_fg_psi` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pnfft_real_f(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pnfft_real_f(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pnfft_real_f` (optional, default = 0).

9.4.2. PFFT-specific Functions

- `FCSResult fcs_p2nfft_set_pfft_patience(`
  `    FCS handle, fcs_int pfft_patience_flag);`
  `FCSResult fcs_p2nfft_get_pfft_patience(`
  `    FCS handle, fcs_int* pfft_patience_flag);`
  `FCSResult fcs_p2nfft_set_pfft_patience_by_name(`
  `    FCS handle, char* patience_name );`

  Set/retrieve flag `pfft_patience` by number or name (optional, default = 1). Feasible values are 0 (`estimate`), 1 (`measure`), 2 (`patient`) and 3 (`exhaustive`).

- `FCSResult fcs_p2nfft_set_pfft_tune(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pfft_tune(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pfft_tune` (optional, default = 0).

- `FCSResult fcs_p2nfft_set_pfft_preserve_input(`
  `    FCS handle, fcs_int yes_or_no);`
  `FCSResult fcs_p2nfft_get_pfft_preserve_input(`
  `    FCS handle, fcs_int* yes_or_no);`

  Set/retrieve flag `pfft_preserve_input` (optional, default = 0).

10. P3M – Particle-Particle Particle-Mesh Ewald

10.1. Features

Periodicity: Only fully periodic boundaries are supported.

Box shape: Triclinic boxes are supported by the analytical differentiation scheme. Any orthorhombic box shape is supported by the analytical and the ik differentiation scheme.

Autotuning: The parameters can be automatically tuned when a required tolerance in the absolute rms field error is provided.

Delegate near-field: Yes.

Virial: Not yet supported, but in progress.

10.2. Solver-specific Parameters

- `tolerance_field` The allowed tolerance of the absolute rms error in the fields. If the required tolerance can't be achieved with the given parameters, the tuning is aborted. Feasible values depend on the system that is computed. In an MD simulation with a thermostat, this should be about an order of magnitude less than the forces generated by the thermostat.

- `r_cut` Cutoff distance of the near field. Can be automatically tuned. Feasible values are in the order of the mean free path between charges. `r_cut` has to be less than half the smallest simulation box length. When `r_cut` is chosen too small, the errors of the algorithm become large, the required accuracy might not be achieved, or the grid has to be chosen very large, which will slow down the algorithm. When `r_cut` is chosen too large, the algorithm might become slow as most computation is done in the badly scaling near field region.

- `grid` Size of the grid. Can be automatically tuned. Feasible values are in the order of $N^{\frac{1}{3}}$, *i.e.* a grid point for each particle. The minimal grid size is 4, the maximal grid size is 512. The larger the grid, the smaller the error, but also the higher the memory and computational requirements of the algorithm.

- `cao` “Charge assignment order”: The number of points in each direction that the charge gets smeared out to. Can be automatically tuned. Allowed values are between 1 and 7. The larger `cao`, the smaller the error, but also the higher the computational cost of the algorithm.
- `alpha` Ewald splitting parameter. Should be automatically tuned. Set this manually only when you know what you are doing.

11. PEPC – Pretty Efficient Parallel Coulomb Solver

Implementation of a highly scalable parallel Barnes-Hut-Treecode for open and (mixed) periodic boundary conditions

The oct-tree method was originally introduced by Josh Barnes and Piet Hut in the mid 1980s to speed up astrophysical N-body simulations with long range interactions[PEPC-1]. Their idea was to use successively larger multipole-groupings of distant particles to reduce the computational effort in the force calculation from the usual $\mathcal{O}(N^2)$ operations needed for brute-force summation to a more amenable $\mathcal{O}(N \log N)$. Though mathematically less elegant than the Fast Multipole Method (see Section 4), the Barnes-Hut algorithm is well suited to dynamic, nonlinear problems and can be combined with multiple-timestep integrators.

The PEPC project[PEPC-2],[PEPC-3] (Pretty Efficient Parallel Coulomb Solver) is a public tree code that has been developed at Jülich Supercomputing Centre since the early 2000s. It is a non-recursive version of the Barnes-Hut algorithm, using a level-by-level approach to both tree construction and traversals.

The parallel version is a hybrid MPI/PThreads implementation of the Warren-Salmon 'Hashed Oct-Tree' scheme, including several variations of the tree traversal routine - the most challenging component in terms of scalability.

Common capabilities

Periodicity: Open, periodic, or mixed boundaries are supported. The box shape is not limited by the chosen periodicity. The potential and field contribution of periodic boundaries are computed using a fast converging renormalization approach borrowed from the FMM, see [PEPC-4] for details.

Box shape: Any (triclinic) box shape is supported.

Tolerances: Not supported.

Delegate near-field: Not supported.

Virial: Not supported.

Additional capabilities

Potential: Internally, PEPC uses a Plummer potential $\frac{1}{\sqrt{r^2+\varepsilon^2}}$, which in the case $\varepsilon = 0$ is identical to the Coulomb potential $\frac{1}{r}$. Using $\varepsilon > 0$, the pole at $r \to 0$ can be smoothed to reduce numerical heating during dynamic simulations.

The value of ε can be modified by setting the method-specific parameter `pepc_epsilon`.

Load Balancing: PEPC includes a sophisticated load balancing scheme, that uses the number of interactions per particle from a previous timestep to estimate and distribute the expected work in the current computation. This can improve the algorithms performance by more than 10%. *Important:* for the load balancing to work correctly, it is necessary that the frontend application does not reorder or redistribute the particles between different calls to `fcs_run()`.

Multithreading: PEPC can employ multiple threads to perform calculations. This can help consolidate mulitple MPI ranks into one rank running multiple threads which will decrease communication volume and memory requirements. In addition to these worker threads and the controlling main thread, PEPC also spawns a background thread that handles the fetching and sending of multipole moments.

On a Blue Gene/Q, one MPI rank per node running sixty worker threads provides good results. PEPC will statically place the main and communications thread on the first of the sixteen cores of the processor while spreading the worker threads on the remaining cores in a round robin fashion.

On JuRoPa, two MPI ranks per node and `pepc_num_walk_threads=7` are optimal. Ensure that `tpt=8` is set in your jobscript on JuRoPa.

Solver-specific parameters

`pepc_theta`: Barnes-Hut Multipole acceptance parameter. Smaller values for `pepc_theta` lead to higher precision but also longer runtime. A value of `pepc_theta=0.0` corresponds to the direct $\mathcal{O}(N^2)$ summation, usual values are in the range of `0.1` $<$ `pepc_theta` $\leq$ `0.6`.

Data type:	`fcs_float`
Default value:	`pepc_theta = 0.6`
Value range value:	`pepc_theta` $\geq$ `0.0`

`pepc_epsilon`: Cutoff parameter of internally used Plummer potential $\frac{1}{\sqrt{r^2+\varepsilon^2}}$, see above.

Data type:	`fcs_float`
Default value:	`pepc_epsilon = 0.0`, i.e. Coulomb interaction.
Value range value:	`pepc_epsilon` $\geq$ `0.0`

`pepc_num_walk_threads`: Number of threads to use during force computation in addition to the communicator and main thread. This should usually correspond the the number of available compute cores per MPI rank.

Data type:	`fcs_int`
Default value:	`pepc_num_walk_threads = 3`
Value range value:	`pepc_num_walk_threads` $\geq$ `1`

`pepc_dipole_correction`: Integer flag to select the extrinsic-to-intrinsic dipole correction for periodic systems.

Possible values for `pepc_dipole_correction`:

0 no extrinsic-to-intrinsic correction
1 *default value:* correction expression as given by Redlack and Grindlay (only for cubic boxes), see [PEPC-5], eqns (19, 20) and [PEPC-6], eq. (5)
2 fictitious charges as given by Kudin (should work for all unit cell shapes), see [PEPC-7], eq. (2.8)
3 measurement of correction value as given in [PEPC-7], eqns. (2.6, 2.7); *currently not implemented*

Data type:	`fcs_int`
Default value:	`pepc_dipole_correction = 1`
Value range value:	`pepc_dipole_correction` $\geq$ `0`

`pepc_load_balancing`: If `pepc_load_balancing` $=$ `0`, the load balancing scheme is disabled, otherwise it is enabled.

Data type:	`fcs_int`
Default value:	`pepc_load_balancing = 0`
Value range value:	`pepc_load_balancing` $\geq$ `0`

`pepc_debug_level`: Solver debug flag mask. Setting zeroth bit, i. e. an odd value activates the solver's status output. Other bits activate different debugging output. See `module_debug.f90` for details.

Data type:	`fcs_int`
Default value:	`pepc_debug_level = 0`
Value range value:	`pepc_debug_level` ≥ 0

`pepc_npm`: Determines the amount of memory allocated for storing tree nodes locally. Two modes can be chosen based on the sign of `pepc_npm`:

- if `pepc_npm` < 0, allocate memory for $10000 \times |$`pepc_npm`$|$ tree nodes,
- if `pepc_npm` > 0, guess the necessary amount N_n and allocate `pepc_npm` $\times N_\mathrm{n}$.

Data type:	`fcs_float`
Default value:	`pepc_npm = -45`
Value range value:	$-\infty <$ `pepc_npm` $< \infty$

Solver specific functions

-
  ```
  fcs_pepc_set_epsilon(FCS handle, fcs_float epsilon)
  fcs_pepc_get_epsilon(FCS handle, fcs_float* epsilon)
  ```

  Set/Retrieve the value for `pepc_epsilon`.

-
  ```
  fcs_pepc_set_theta(FCS handle, fcs_float theta)
  fcs_pepc_get_theta(FCS handle, fcs_float* theta)
  ```

  Set/Retrieve the value for `pepc_theta`.

-
  ```
  fcs_pepc_set_debuglevel(FCS handle, fcs_int level)
  fcs_pepc_get_debuglevel(FCS handle, fcs_int* level)
  ```

  Set/Retrieve the value for PEPCs internal debug-level bitmask. This is only intended for development purposes. See file `lib/pepc/src/module_debug.f90` for details.

-
  ```
  fcs_pepc_set_num_walk_threads(FCS handle, fcs_int num_walk_threads)
  fcs_pepc_get_num_walk_threads(FCS handle, fcs_int* num_walk_threads)
  ```

  Set/Retrieve the value for `pepc_num_walk_threads`.

-
  ```
  fcs_pepc_set_load_balancing(FCS handle, fcs_int load_balancing)
  fcs_pepc_get_load_balancing(FCS handle, fcs_int* load_balancing)
  ```

  Set/Retrieve the value for `pepc_load_balancing`.

-
  ```
  fcs_pepc_set_dipole_correction(FCS handle, fcs_int dipole_correction)
  fcs_pepc_get_dipole_correction(FCS handle, fcs_int* dipole_correction)
  ```

  Set/Retrieve the value for `pepc_dipole_correction`.

-
  ```
  fcs_pepc_set_npm(FCS handle, fcs_float npm)
  fcs_pepc_set_npm(FCS handle, fcs_float* npm)
  ```

  Set/Retrieve the value for `pepc_npm`.

Known bugs or missing features

- virial is not computed at all in PEPC (functionality was lost during transition from old version to pepc-2.0)

- 1D-, 2D-periodicity still needs to be tested.
- Warns that non-cubic systems are experimental.
  The following requirements must be satisfied in any case:
  - box must be approximately rectangular
  - box edges in periodic directions must not be significantly shorter than the longest edge

quantify these requirements

warn only once on one process, if box is non-cubic

- How to compute the virial for periodic boundaries?

References

PEPC-1 Nature **324**, 446 (1986), `http://dx.doi.org/10.1038/324446a0`

PEPC-2 CPC **183**, 880–889, `http://dx.doi.org/10.1016/j.cpc.2011.12.013`

PEPC-3 PEPC web page, `http://www.fz-juelich.de/ias/jsc/pepc`

PEPC-4 J. Chem. Phys. **121**, 2886 (2004), `http://link.aip.org/link/doi/10.1063/1.1771634`

PEPC-5 J. Chem. Phys. **107**, 10131 (1997), `http://link.aip.org/link/doi/10.1063/1.474150`

PEPC-6 J. Chem. Phys. **101**, 5024 (1994), `http://link.aip.org/link/doi/10.1063/1.467425`

PEPC-7 Chem. Phys. Lett. **283**, 61 (1998), `http://dx.doi.org/10.1016/S0009-2614(98)00468-0`

move PEPC citations to bibliography

12. PP3MG – NameExpanded

PP3MG implements a multigrid method solver. The particle charges are interpolated to a regular grid. The long-range part is then solved via solving the Poisson equation, using finite difference/finite volume discretization. The short-range part is computed directly.

```
Parameters:

  - too many, too poorly documented
    rather use meaningful defaults, or values already known in the handle

Box Shape / Periodicity

  - open or mixed boundaries:  is this supported? for what boxes?
  - periodic boundaries:       only rectangular or cubic boxes?

Can Delegate Nearfield to MD:  currently not

Bugs / Missing Features:

  - remove superfluous parameters like periodicity or mpi_dims;
    these are already known in the handle
  - provide meaningful defaults for parameters like mesh sizes or
    workspace size
  - document what the remaining parameters mean, and give rules
    how to choose reasonable values
  - crashes - doesn't appear to work at present, except for 8 atom
    sample in unit cube
  - excessive use of pow function - this is slow
  - should use solution of previos MD step as start for the iteration
```

13. vmg – Versatile Multigrid

vmg is a grid-based solver for computing the long-range Coulomb interactions of particles for periodic boundary conditions.

In depth, the Coulomb problem is first split into a short-range and a long-range part, for when treated individually each can be solved efficiently. This is done by adding so-called shield charges at the same position as each point-like charge, see figure 13.1. These are described by a spline function that has compact support, is symmectric around its center and is normalized to the same magnitude as the original point-like charge. The short-range part (center, fig. 13.1) is then solved by a direct summation of the pair-wise interactions between the small number of charges lying in the compact support of these shield charges. Outside of the support, shield charge and oppositely charged point-like charge precisely cancel. The long-range part consists of the potential induced by these shield charges. It is treated by solving a well-known partial differential equation (PDE) in the form $Lu = f$, the Poisson equation, where L is the negated Laplace operator $L := -\Delta$, u is the sought-for solution and f is the "right-hand side", i. e. the shield charges sampled on the grid.

Solving PDEs numerically is usually done by discretizing their differential operator in an extension of the idea of calculating a derivative via the method of finite differences. This converts the continuous problem $Lu = f$ into a discrete problem $Ax = b$, where A is the matrix of the discretized operator, x is the discrete solution and b is again the "right-hand side". In the multigrid ansatz, see [VMG-2], $Ax = b$ is obtained on the finest grid (of d dimensions) while on coarser grids the so-called defect equation $Ax' = b - Ax$ is solved. This offers the advantage that the solution time is independent of the system size. The basic idea of a multigrid is to eliminate the residual, i. e. the error $(b - Ax)$, at each coarser and coarser level. The picture to have in mind is that the "noise" is reduced at multiple frequencies at the same time.

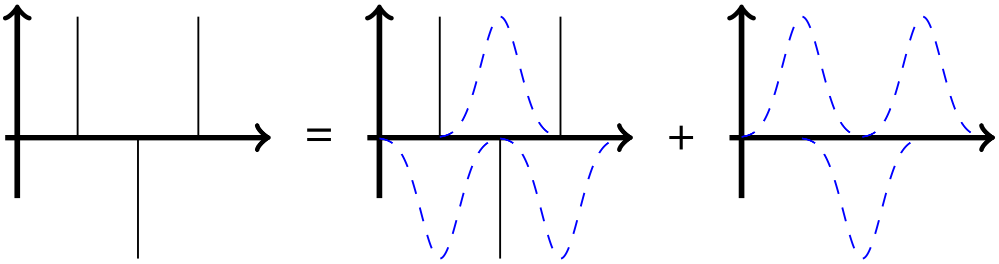

Figure 13.1.: The charge distribution consisting of point charges (left) is split into a smoothened part (right) and the rest (center), see [VMG-1]

A cycle of two grids, i. e. a fine grid and a coarser one, then consists of steps of "removing noise" (smoothing), projecting down onto the coarser grid (restriction), solving exactly and bringing this solution back onto the finer grid (prolongation). A multigrid cycle then consists of the recursive application of two-grid cycles instead of solving exactly, i. e. we step down the grid hierarchy from a finer to the next coarser grid and eventually move up again.

There are many types of this cycle differing in the precise sequence of smoothing, prolongation, and restriction steps. Since this is an iterative process we have to perform such a cycle a (typically small) number of times until the residual is small enough. Then we have "converged" sufficiently to the solution.

In order to set up the above PDE, we need the right-hand side b that consists of the aforementioned shield charges. After solving the PDE on the grid, the Coulomb potential at the charge positions is obtained by some interpolation scheme. Right now, a multidimensional Newton interpolation is used, see [VMG-3], where a polynomial is fit to the solution around the desired position. The field is calculated via analytical differentiation of these polynomials.

To sum it all up, the multigrid does the following:

1. Bring the shield charges onto the grid (the right-hand side) by evaluating spline functions for each particle charge.
2. Iterate over the multigrid in a specific cycle until either the (absolute or relative) residual is lower than a given threshold value (converged) or a set maximum number of iterations steps is reached (non-converged).
3. The solution is interpolated back at the position of each charge, yielding the long-range interactions.
4. Short-range interactions are determined via direct summation.

Crucial for its accuracy are then the following parameters of the multigrid solver:

grid size This determines the number of grid points per axis on the finest level. This parameter mainly affects the accuracy of the long-range part of the solution.

width of spline support This gives the width of the shield charges. The greater is the width, the more accurate the solution becomes, but at the same time the more interactions have to be evaluated with the (computationally expensive) short-range part of the solver. One has to ensure that the number of particles in the support of these shield charges is kept constant, otherwise the algorithm does not scale optimally anymore.

discretization scheme This is the manner of constructing a matrix A from the continuous Laplace operator L. A scheme of higher order gives greater accuracy w.r.t. a fixed grid size. This parameter is solely affecting the accuracy of the long-range part of the algorithm.

For parallelization the particles are distributed over many processes. Each brings its local particles onto the grid. Also, each process has a certain share of the grid and performs its multigrid operations on its share, communicating with neighboring processes during the restriction, prolongation, and smoothing. Conceptionally, there is also one global communication in every multigrid iteration where the local residuals get summed up globally. Eventually, each process interpolates back its particle potential.

Common capabilities

Periodicity: Currently only periodic boundary conditions are supported. Support for open or mixed boundary conditions is in progress.

Box shape: Cubic boxes are supported.

Tolerances: ?

Delegate near-field: Not implemented.

Virial: No.

Additional capabilities

Due to the aforementioned splitting of the Coulomb interactions into a short- and a long-range part, some parameters affect only the accuracy of either and not necessarily of both, see their desription. If then a system only has a small long-range part, one may save substantial calculation time by decreasing accuracy specifically for long-range calculations. This allows checking whether either interactions part far outweighs the other. Via the configure switch *–enable-debug-output* the various energy contributions of short- and long-range parts are printed to the console. Note however that this switch should *not be used for production runs* as it makes unnecessary additional calculations once an efficient set of parameters has been found.

Solver specific functions

- `fcs_vmg_set_max_level(FCS handle, fcs_int max_level);`

  Sets the number of points per axis of the finest grid (optional,default=6). More specifically it is $2^{\text{max\_level}}$, i.e. more points means better approximated solution but scales with $\mathcal{O}(\text{max\_level}^3)$. This is the first value to change for better accuracy, see also near_field_cells.

- `fcs_vmg_get_max_level(FCS handle, fcs_int* max_level);`

  Get the maximum level of the multigrid algorithm.

- `fcs_vmg_set_max_iterations(FCS handle, fcs_int max_iterations);`

  Sets the maximum number of multigrid iterations (optional, default=15). This is the second threshold besides *precision* that will stop the iteration. Normally, you don't need to change this value, this is only present in case the iteration does not converge and this sets after how many iteration steps it is considered as "not converged".

- `fcs_vmg_get_max_iterations(FCS handle, fcs_int* max_iterations);`

  Get the maximum number of multigrid iterations.

- `fcs_vmg_set_smoothing_steps(FCS handle, fcs_int smoothing_steps);`

  Set the number of smoothing steps in the multigrid cycle per level (optional, default=3). Smoothing refers to removal of error at a certain level. This is usually done for a fixed and small number of iterations. Usually, there is no need to change it. Fewer steps may be faster (and may cause solution not to converge anymore). Does not affect the precision but only the computational efficiency.

- `fcs_vmg_get_smoothing_steps(FCS handle, fcs_int *smoothing_steps);`

  Get the number of pre/post-smoothing steps on each level.

- `fcs_vmg_set_cycle_type(FCS handle, fcs_int cycle_type);`

  Specifies the type of multigrid cycle, 1 V-cycle, 2 W-cycle, ... (optional, default=1). For parallel computation, V-cycle is the correct type, for serial computation W-cycle might be slightly more efficient. In case of doubt, leave it to default.

- `fcs_vmg_get_cycle_type(FCS handle, fcs_int *cycle_type);`

  Get the cycle_type-number of the multigrid cycle used.

- `fcs_vmg_set_precision(FCS handle, fcs_float precision);`

  Set the threshold for (absolute and relative) residual below which internal iteration is stopped (optional, default=1.0e-8). This is not equal to FCS's global tolerance! As an estimate one might use one magnitude below desired tolerance.

- `fcs_vmg_get_precision(FCS handle, fcs_float *precision);`

- `fcs_vmg_set_near_field_cells(FCS handle, fcs_int near_field_cells);`

  Set the half witdh in number of grid points of compact local support of b-spline function (radius of shielding charges) (optional, default=4). This value along with max_level has great impact on accuracy but the correlation is reciprocal.

- `fcs_vmg_get_near_field_cells(FCS handle, fcs_int *near_field_cells);`

  Get the number of near field cells for separating the near/far field part of the potential.

- `fcs_vmg_set_interpolation_order(FCS handle, fcs_int interpolation_order);`

  Set polynomial order of tensored Newton back-interpolation of solution at the site of each charge (optional, default=5). Usually, does not need to be changed. Normally, less means precision of grid solution is omitted, more means useless computation.

- `fcs_vmg_get_interpolation_order(FCS handle, fcs_int *interpolation_order);`

  Get the interpolation order for interpolating the gridded potential to the particle positions.

- `fcs_vmg_set_discretization_order(FCS handle, fcs_int discretization_order);`

  Discretization scheme in setting up partial differential equation (optional, [2,4], default=4). This heavily influences precision of solution in conjunction with near_field_cells ans max_level. This parameter enables direct control of the discretization error of the long-range part. Default is 4 and leave it at that.

- `fcs_vmg_get_discretization_order(FCS handle, fcs_int *discretization_order);`

  Get the order of the discretization scheme.

Known bugs or missing features

- Open and mixed boundaries still need to be incorporated.
- Separate near-field part?
- How to compute the virial?

References

move vmg citations to bibliography

vmg-1 M. Griebel, S. Knapek, G. Zumbusch **Numerical Simulation in Molecular Dynamics – Numerics, Algorithms, Parallelization, Applications**, 2007

vmg-2 U. Trottenberg, C. W. Oosterlee, A. Schuller **Multigrid**, 2000

vmg-3 K. A. Atkinson **Introduction to Numerical Analysis**, 1988

14. direct – Direct summation

The direct solver implemented within the ScaFaCoS library performs a direct summation of the pair-wise interactions between all given particles. The parallel computations use the given distribution of particles among parallel MPI processes (i.e., no additional redistribution for improving the load-balancing is performed). Let p be the number of parallel MPI processes. After each process has computed the interactions between its local particles, $p - 1$ steps are performed to compute interactions with all non-local particles. In each step, each process receives the particles of the preceding process, computes the interactions with its local particles, and sends to the previously received particles to the succeeding process.

Common capabilities

Periodicity: Open, periodic, or mixed boundaries are supported. The box shape is not limited by the chosen periodicity. Periodic boundaries are computed by placing a number of periodic images of the particle system around the given particle system. The number of images to use in each (periodic) dimensions can be specified with `fcs_direct_set_periodic_images`.

Box shape: Any (triclinic) box shape is supported.

Tolerances: No.

Delegate near-field: No.

Virial: ?

Additional capabilities

Cutoff: `fcs_direct_set_cutoff` can be used to specify a cutoff range that limits the computation of interactions. If the cutoff range is 0, then all interactions are considered. If the cutoff range is greater than 0, then only interactions inside the cufoff range are considered. If the cutoff range is less than 0, then only interactions outside the (positive) cufoff range are considered. If the cutoff range is greater than 0, then `fcs_direct_set_cutoff_with_near` can be used to enable the ScaFaCoS internal near-field solver module instead of the direct solver.

Solver specific functions

- `fcs_direct_set_cutoff(FCS handle, fcs_float cutoff);`

  Set the current cutoff range (optional, default = 0).

- `fcs_direct_get_cutoff(FCS handle, fcs_float *cutoff);`

  Retrieve the current cutoff range.

- `fcs_direct_set_periodic_images(FCS handle, fcs_int *periodic_images);`

  Set the number of periodic images to use for computations with periodic boundaries (optional, default = { 1, 1, 1 }).

- `fcs_direct_get_periodic_images(FCS handle, fcs_int *periodic_images);`

  Retrieve the number of periodic images used for computations with periodic boundaries.

- `fcs_direct_set_cutoff_with_near(FCS handle, fcs_bool cutoff_with_near);`

  Enable the near-field solver module (instead of the direct solver) to be used for computations with a cutoff range.

- `fcs_direct_get_cutoff_with_near(FCS handle, fcs_bool *cutoff_with_near);`

  Retrieve whether the near-field solver module is used for computations with a cutoff range.

Known bugs or missing features

- Periodic boundaries still need to be tested.
- How to compute the virial for periodic boundaries?

15. List of Functions

This chapter lists all the functions within the ScaFaCoS interface, which can be used by the user to manipulate the library.

15.1. Mandatory Functions

```
/* C */
FCSResult fcs_init(FCS handle, char* method, MPI_Comm comm);
```

```
! Fortran
function fcs_init(handle,method,comm)
type(c_ptr)                                    :: handle
char                                           :: method(*)
integer(kind = c_int)                          :: comm
type(c_ptr)                                    :: fcs_init
```

Figure 15.1.: Function call: fcs_init

```
/* C */
FCSResult fcs_set_common(FCS handle, fcs_int ↩
   short_range_flag, fcs_float* box_a, fcs_float* box_b, ↩
   fcs_float* box_c, fcs_float* offset, fcs_int* ↩
   periodicity, fcs_int total_particles);
```

```
! Fortran
function fcs_set_common(handle,short_range_flag,box_a,↩
   box_b,box_c,offset,periodicity,total_particles)
type(c_ptr)                                   ::  handle
logical                                       ::  ↩
   short_range_flag
real(kind = fcs_real_kind_isoc)               ::  box_a(3)
real(kind = fcs_real_kind_isoc)               ::  box_b(3)
real(kind = fcs_real_kind_isoc)               ::  box_c(3)
real(kind = fcs_real_kind_isoc)               ::  offset(3)
logical                                       ::  ↩
   periodicity(3)
integer(kind = fcs_integer_kind_isoc)         ::  ↩
   total_parts
type(c_ptr)                                   ::  ↩
   fcs_set_common
```

Figure 15.2.: Function call: fcs_common_set

```
/* C */
FCSResult fcs_tune (FCS handle, fcs_int local_particles, ↩
   fcs_float *positions, fcs_float *charges);
```

```
! Fortran
function fcs_tune(handle,n_locp,n_maxlocp,positions,↩
   charges)
type(c_ptr), value                            :: handle
integer(kind = fcs_integer_kind_isoc),value :: n_locp
real(kind = fcs_real_kind_isoc)               :: positions↩
   (3*n_locp)
real(kind = fcs_real_kind_isoc)               :: charges(↩
   n_locp)
type(c_ptr)                                   :: fcs_tune
```

Figure 15.3.: Function call: fcs_tune

```
/* C */
FCSResult fcs_run (FCS handle, fcs_int local_particles, ↩
   fcs_float *positions, fcs_float *charges, fcs_float *↩
   field, fcs_float *potentials);
```

```
! Fortran
function fcs_run(handle,n_locp,n_maxlocp,positions,charges↩
   ,field,potential)
type(c_ptr), value                              :: handle
integer(kind = fcs_integer_kind_isoc),value :: n_locp
real(kind = fcs_real_kind_isoc)                 :: positions↩
   (3*n_locp)
real(kind = fcs_real_kind_isoc)                 :: charges(↩
   n_locp)
real(kind = fcs_real_kind_isoc)                 :: field(3*↩
   n_locp)
real(kind = fcs_real_kind_isoc)                 :: potential(↩
   n_locp)
type(c_ptr)                                     :: fcs_run
```

Figure 15.4.: Function call: fcs_run

```
/* C */
FCSResult fcs_destroy (FCS handle);
```

```
! Fortran
function fcs_destroy(handle)
type(c_ptr), value                              :: handle
type(c_ptr)                                     :: fcs_run
```

Figure 15.5.: Function call: fcs_destroy

15.2. Generic Functions

15.2.1. Errorhandling

```
/* C */
fcs_int fcsResult_getReturnCode (FCSResult err);
```

```
! Fortran
function fcsResult_getReturnCode(res)
type(c_ptr)                             ::  res
integer(kind = fcs_integer_kind_isoc)   ::  ↩
   fcsResult_getReturnCode
```

Figure 15.6.: Function call: fcsResult_getReturnCode

```
/* C */
const char* fcsResult_getErrorMessage (FCSResult err);
```

```
! Fortran
function fcsResult_getErrorMessage(res)
type(c_ptr)                                       ::  res
character(kind = c_char, len = MESSAGE_LENGTH)    ::  ↩
   fcsResult_getErrorMessage
```

Figure 15.7.: Function call: fcsResult_getErrorMessage

```
/* C */
const char* fcsResult_getErrorSource (FCSResult err);
```

```
! Fortran
function fcsResult_getErrorSource(res)
type(c_ptr)                                       ::  res
character(kind = c_char, len = MESSAGE_LENGTH)    ::  ↩
   fcsResult_getErrorSource
```

Figure 15.8.: Function call: fcsResult_getErrorSource

15.2.2. Getters and Setters

15.2.3. Near Field Computations

```
/* C */
FCSResult fcs_compute_near_field (FCS handle, fcs_float ↩
   dist, fcs_float *field);
```

```
! Fortran
function fcs_compute_near_field(handle,dist,field)
type(c_ptr)                                      ::  handle
real(kind = fcs_real_kind_isoc)                  ::  dist
real(kind = fcs_real_kind_isoc), dimension(3)    ::  field
type(c_ptr)                                      ::  ↩
   fcs_compute_near_field
```

Figure 15.9.: Function call: fcs_compute_near_field

```
/* C */
FCSResult fcs_compute_near_potential (FCS handle, ↩
   fcs_float dist, fcs_float *pot);
```

```
! Fortran
function fcs_compute_near_potential(handle,dist,pot)
type(c_ptr)                                      ::  handle
real(kind = fcs_real_kind_isoc)                  ::  dist
real(kind = fcs_real_kind_isoc)                  ::  pot
type(c_ptr)                                      ::  ↩
   fcs_compute_near_potential
```

Figure 15.10.: Function call: fcs_compute_near_potential

fcs_method_has_nea
fehlt!

```
/* C */
FCSResult fcs_compute_near (FCS handle, fcs_float dist, ↩
   fcs_float *pot, fcs_float *field);
```

```
! Fortran
function fcs_compute_near(handle,dist,pot,field)
type(c_ptr)                                      ::  handle
real(kind = fcs_real_kind_isoc)                  ::  dist
real(kind = fcs_real_kind_isoc)                  ::  pot
real(kind = fcs_real_kind_isoc), dimension(3)    ::  field
type(c_ptr)                                      ::  ↩
   fcs_compute_near
```

Figure 15.11.: Function call: fcs_compute_near

```
/* C */
FCSResult fcs_method_has_near (FCS handle, fcs_int *↩
   has_near);
```

```
! Fortran
function fcs_method_has_near(handle,has_near)
type(c_ptr)                                      ::  handle
real(kind = fcs_real_kind_isoc)                  ::  dist
real(kind = fcs_real_kind_isoc), dimension(3)    ::  field
type(c_ptr)                                      ::  ↩
   fcs_compute_near_field
```

Figure 15.12.: Function call: fcs_method_has_near

15.2.4. Output

```
/* C */
void fcs_printHandle (FCS handle);
```

```
! Fortran
subroutine fcs_printHandle(handle)
type(c_ptr)                                   ::  handle
```

Figure 15.13.: Function call: fcs_printHandle

15.2.5. Parser

```
/* C */
FCSResult fcs_parser (FCS handle, const char* parse_string↩
   );
```

```
! Fortran
function fcs_printHandle(handle,parse_string)
type(c_ptr)                                   ::  handle
character(kind = c_char, len = *)             ::  ↩
   parse_string
type(c_ptr)                                   ::  ↩
   fcs_parser
```

Figure 15.14.: Function call: fcs_parser

15.2.6. Virial

```
/* C */
FCSResult fcs_require_virial (FCS handle, fcs_int flag);
```

```
! Fortran
function fcs_require_virial(handle,flag)
type(c_ptr)                                   ::  handle
logical                                       ::  dist
type(c_ptr)                                   ::  ↩
   fcs_require_virial
```

Figure 15.15.: Function call: fcs_require_virial

```
/* C */
FCSResult fcs_get_virial (FCS handle, fcs_float* virial);
```

```
! Fortran
function fcs_get_virial(handle,virial)
type(c_ptr)                                     ::  handle
real(kind = fcs_real_kind_isoc), dimension(9)   ::  dist
type(c_ptr)                                     ::  ↩
   fcs_require_virial
```

Figure 15.16.: Function call: fcs_get_virial

15.3. Direct Solver specific Functions

15.3.1. Getters and Setters

15.4. Ewald Solver specific Functions

15.4.1. Getters and Setters

15.5. FMM Solver specific Functions

15.5.1. Getters and Setters

15.6. MEMD Solver specific Functions

15.6.1. Getters and Setters

15.7. MMM1D Solver specific Functions

15.7.1. Getters and Setters

15.8. MMM2D Solver specific Functions

15.8.1. Getters and Setters

15.9. PEPC Solver specific Functions

15.9.1. Getters and Setters

15.10. PP3MG Solver specific Functions

15.10.1. Getters and Setters

15.11. P^{2}NFFT Solver specific Functions

15.11.1. Getters and Setters

15.12. P^{3}M Solver specific Functions

15.12.1. Getters and Setters

15.12.2. Near Field Computations

```
/* C */
FCSResult fcs_p3m_compute_near_field (FCS handle, ↩
   fcs_float dist, fcs_float *field);
```

```
! Fortran
function fcs_p3m_compute_near_field(handle,dist,field)
type(c_ptr)                                      ::  handle
real(kind = fcs_real_kind_isoc)                  ::  dist
real(kind = fcs_real_kind_isoc), dimension(3)    ::  field
type(c_ptr)                                      ::  ↩
   fcs_p3m_compute_near_field
```

Figure 15.17.: Function call: fcs_p3m_compute_near_field

```
/* C */
FCSResult fcs_p3m_compute_near_potential (FCS handle, ↩
   fcs_float dist, fcs_float *pot);
```

```
! Fortran
function fcs_p3m_compute_near_potential(handle,dist,pot)
type(c_ptr)                                      ::  handle
real(kind = fcs_real_kind_isoc)                  ::  dist
real(kind = fcs_real_kind_isoc)                  ::  pot
type(c_ptr)                                      ::  ↩
   fcs_p3m_compute_near_potential
```

Figure 15.18.: Function call: fcs_p3m_compute_near_potential

```
/* C */
FCSResult fcs_p3m_compute_near (FCS handle, fcs_float dist↩
   , fcs_float *pot, fcs_float *field);
```

```
! Fortran
function fcs_p3m_compute_near(handle,dist,pot,field)
type(c_ptr)                                      ::  handle
real(kind = fcs_real_kind_isoc)                  ::  dist
real(kind = fcs_real_kind_isoc)                  ::  pot
real(kind = fcs_real_kind_isoc), dimension(3)    ::  field
type(c_ptr)                                      ::  ↩
   fcs_p3m_compute_near
```

Figure 15.19.: Function call: fcs_p3m_compute_near

15.13. VMG Solver specific Functions

Part II.

Developer's Guide

16. Build System

```
Prerequisites
-------------

This source tree uses GNU autotools
<http://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html>

In order to do development, please make sure you have reasonably recent
versions of the following tools installed and in your $PATH:
  GNU m4       >= 1.4.13  http://ftp.gnu.org/gnu/m4/m4-1.4.16.tar.gz
  GNU Autoconf >= 2.64    http://ftp.gnu.org/gnu/autoconf/autoconf-2.68.tar.gz
  GNU Automake >= 1.11    http://ftp.gnu.org/gnu/automake/automake-1.11.3.tar.gz
  GNU Libtool  >= 2.2.6b  http://ftp.gnu.org/gnu/libtool/libtool-2.4.2.tar.gz

Then, set up the autotools infrastructure:
  ./bootstrap

This should create configure, Makefile.in, and config.h.in files.

If that went without trouble, continue as described in the README file.
```

To learn how to install the Autotools, have a look at http://www.gnu.org/software/automake/faq/autotools-faq.html#How-do-I-install-the-Autotools-_0028as-user_0029_003f.

```
Build system layout
-------------------

There is a toplevel configure.ac script and helper macros in the m4/
subdirectory.  Each directory where stuff needs to be done later gets
a Makefile.am file which is processed by automake and later by configure.

In order to facilitate modularity of the different solver methods, each
method gets its own configure.ac script in libs/$method/ as well.  The
toplevel configure will call each of the lower-level ones which are
enabled and present in turn.

The generated makefiles support all of the standard targets described here:
```

<http://www.gnu.org/software/automake/manual/html_node/Standard-Targets.html>

Generated files

The autotools build system uses several generated files and a few helper scripts:

1) Helper scripts installed below build-aux/ subdirectories:
- depcomp
- install-sh
- compile
- missing

2a) File generated at bootstrap time, and distributed to end-users:

- aclocal.m4	macro files generated by aclocal
- configure	scripts are generated from configure.ac files and additional macro files (aclocal.m4 and files in some m4/ subdirectory)
- Makefile.in	template files generated by automake (and later converted to Makefile files by config.status)
- config.h.in	header template file generated by autoheader (and later converted to config.h by config.status)

2b) File generated at bootstrap time, NOT distributed to end-users:

- autom4te.cache	directory containing autotools-internal cache files. This may be safely removed at any time.

3) Files generated at configure run time by the end-user:

- config.log	a log file containing detailed configure test results
- config.status	a script file containing the test results
- config.cache	a cache of test results (used when --config-cache is given)
- Makefile	the actual makefiles generated by config.status
- config.h	a project-specific header that should not be installed
- .deps/*	dependency tracking makefile snippets
- stamp-*	makefile stamp files

and of course object files and programs etc.

None of these files should be committed to SVN, because the files in (1) and in (2) may differ between different autotools versions (causing spurious differences with "svn diff" when two developers use different versions) and because the files in (3) are system-dependent.

```
For further reading see:
<http://www.gnu.org/software/autoconf/manual/html_node/Making-configure-Scripts.html>
<http://www.gnu.org/software/automake/manual/html_node/CVS.html>

How to add a new solver
-----------------------

If you are adding a new solver to the build system, please adjust the
following build system files:

- Either convert your build system below lib/SOLVER to autotools or ensure
  that the usual GNU targets are supported by your makefiles and that VPATH
  building works; see here for more information:
  <http://www.gnu.org/software/automake/manual/html_node/Third_002dParty-Makefiles.html>
  <http://www.gnu.org/software/automake/manual/html_node/Standard-Targets.html>

- Add your solver to the toplevel all_solver_methods macro in the toplevel
  configure.ac script.  If your solver contains GPL code, or relies on GPL
  libraries, add your solver to the gpl_solver_methods macro in the toplevel
  configure.ac script.  Add the list of libraries the user should link to
  to the SCAFACOS_LIBS variable near the end of the script.

- Document in the toplevel README any required libraries for your solver.

- For each Makefile.am you add,
  - adjust the SUBDIRS line in the next-higher Makefile.am file,
  - add an 'AC_CONFIG_FILES([.../Makefile])' line to the next-higher
    configure.ac, with the correct relative path.

- In each Makefile.am file that deals with preprocessed Fortran, add
    include $(top_srcdir)/build-aux/fortran-rules.am

  to ensure .f90 files are preprocessed.

- For each C, C++ source file, add
    #ifdef HAVE_CONFIG_H
    #include <config.h>
    #endif

  before the first included header.  However, do not add this to header
  files, esp. not to header files that are installed later.
  (Note that end-users of scafacos like IMD should instead include
  an installed header like fcs.h.)

- For preprocessed Fortran source file in a method, add
    #ifdef HAVE_FCONFIG_H
    #include <fconfig.h>
    #endif

  before the first included header.

- In the toplevel fconfig.h.in file, add the following lines for your solver:
    ! Whether solver method <solver> is enabled.
    #undef FCS_ENABLE_<SOLVER>

- Ensure src and test are adjusted (FIXME: please elaborate here)

- Finally, rerun ./bootstrap in the toplevel source directory, then proceed
  as described in README.
```

17. Implementation

17.1. Licenses

17.1.1. Applying the (L)GPL

To apply the (L)GPL to your program, do the following:

- Add the following header text in a comment to all of your files (source files and text files!):

  ```
  Copyright (C) 2011 <authors>

  This file is part of ScaFaCoS.

  ScaFaCoS is free software: you can redistribute it and/or modify it
  under the terms of the GNU Lesser Public License as published by the
  Free Software Foundation, either version 3 of the License, or (at
  your option) any later version.

  ScaFaCoS is distributed in the hope that it will be useful, but
  WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser Public License for more details.

  You should have received a copy of the GNU Lesser Public License
  along with this program.  If not, see
  <http://www.gnu.org/licenses/>.
  ```

- Replace the word `<authors>` with the names of the authors that have written the file. If you want to use the GPL instead of the LGPL, replace the word *Lesser* in the text by *General* in all three places.

17.1.2. Simple files

For simple files, you might prefer not to use the long header of the (L)GPL, instead you can use:

```
Copyright (C) 2011 <authors>
```

Copying and distribution of this file, with or without modification, are permitted in any medium without royalty provided the copyright notice and this notice are preserved. This file is offered as-is, without any warranty.

18. Test Environment

18.1. Generic Test Program `scafacos_test`

The generic test program `scafacos_test` located in `tests/generic` can be used to perform precision and performance tests with all solver methods in a common way. The test case to be computed has to specified in form of an *XML file*. A single XML file can contain several particle configurations to be computed one after another using the same solver method. The particle data (i.e., position and charge values as well as optional potential and field reference values) can either be read from the input file(s) (plain text or binary format) or generated *online* using several internal particle data generators. Furthermore, the particle system(s) to be computed can explicitly increased by duplicating the given input particles several times. Additionally, the test program can write the result of the computations to a new XML output file. This can be used to create potential and field reference values using one of the implemented solvers as reference method.

Usage:

```
./scafacos_test [OPTIONS] METHOD FILE
```

The test program has to be executed in parallel using MPI (e.g. with `mpiexec -n NUMPROCS ./scafacos_test...`). `METHOD` specifies the solver method to be used and can be any method name supported by the FCS init function `fcs_init` (see 3.2.3). Additionally, the method name can also be `none` to pass the test case through the test program (including parsing, particle data generation, duplication, and output), but without executing a solver. `FILE` specifies the XML file describing the test case to be computed. If the zlib compression library was available during building the library, then the XML files can also be compressed. The command line arguments are only read on the master process.

Options:

The given command line arguments override conflicting settings that may already exist within the XML file that specifies the test case.

`-o OUTFILE`	Write the given test case including the computed results to a new XML file `OUTFILE`.
`-b`	Write the particle data in a machine-dependent binary format to a separate file (basename of `OUTFILE` with 'bin' suffix. This is only useful together with `-o` option.
`-d DUP`	Duplicate a given periodic particle system in each *periodic* dimension. `DUP` can be a single value X or three values XxYxZ (i.e., one separate value for each dimension). The default value of `DUP` = 1 is equivalent to no duplication.
`-m MODE`	Specify how the input particle data is distributed among parallel processes before the solver is executed. Mode can be one of the following: • `atomistic`: Equal distribution without any further assumptions (this is the *default*). • `all_on_master`: All particles are on the master process. • `random`: Equal distribution after a random redistribution. • `domain`: Redistribution depending on the particle positions. This represents a regular domain decomposition with respect to a Cartesian grid of MPI processes. The process grid is created by MPI and supplied to the solver in form of a Cartesian communicator.
`-c CONF`	Set additional method configuration parameters. `CONF` is given to FCS function `fcs_parser` (see ??).
`-i ITER`	Perform `ITER` number of runs with each configuration.
`-r RES`	Compute results only for `RES` of the given particles, where `RES` can be an absolute integer number of particles (i.e., without '.') or a fractional number (i.e., with '.') relative to the given particles. The default value `RES`=1.0 is equivalent to all particles. If the direct solver is used, then all other particles (i.e., the particles for which no results are computed) are used as additional input particles for the computations.

Text case XML file format

The XML file contains all necessary information to describe the particle configurations to be computed. The master process reads the whole XML file at once and stores the information of all particle configurations in its local memory. Plain text particle data contained in the XML file is also stored at once on the master process. Input of binary particle data as well as any generation or duplication of particle data is performed by all processes in parallel just before the corresponding particle configuration is prepared for the computations. The root element of the test case XML file is called `scafacos_test`.

Element	`<scafacos_test>...</scafacos_test>`
description	Root element of the test case XML file
Child-elements	`<configuration>*`

Attributes	**Type**	**Description**
`name`	string	Name of the test case.
`description`	string	One-line description of the test case.
`reference_method`	string	Name of the method used to compute the reference potential and field values.
`error_potential`	string	Estimated maximal error of the reference potential values.
`error_field`	string	Estimated maximal error of the reference field values.

Each test case XML file can contain several particle configurations that should be computed one after another using the same solver method. The XML element specifying a particle configuration is call `configuration`. Child-elements are used to add particles in form of plain text or binary particle data as well as through the particle generators or explicit duplication of given particles. There can be an arbitrary number of `particle`, `binary`, and `generate` elements. There can be only one `duplicate` element that specifies the duplication for *all* given particles.

Element	`<configuration>...</configuration>`
Description	Particle configuration to be computed.
Child-elements	`<particle>*`, `<binary>*`, `<generate>*`, `<duplicate>`

Attributes	**Type**	**Description**
`offset`	float[3]	Origin of the system box.
`box_a`	float[3]	First base vector of the system box.
`box_b`	float[3]	Second base vector of the system box.
`box_c`	float[3]	Third base vector of the system box.
`periodicity`	bool[3]	Periodicity of the system.
`epsilon`	float or 'metallic'	Value of the dielectric permittivity of the system boundary. Use 'metallic' for infinite permittivity, i.e. metallic boundary conditions.
`decomposition`	string	Input particle distribution: 'atomistic', 'all_on_master', 'random', or 'domain'. See the description of the `-m` option of the test program for further information about the difference input distributions.

Plain text and binary particle data input

A single particle can be specified as plain text within the XML file using the `particle` element. The attributes of this element are used to specify the position, charge, reference potential, and reference field values. There can be an arbitrary number of these `particle` elements.

Element	`<particle>...</particle>`
Description	Data of a single particle.
Child-elements	

Attributes	**Type**	**Description**
`position`	float[3]	Position of the particle.
`q`	float	Charge of the particle.
`potential`	float	Reference potential at the position of the particle.
`field`	float[3]	Reference field at the position of the particle.

Binary particle data input from a separate file can be specified with the `binary` element. The attributes of this element are used to specify the filename, offset, and total number of particles to read. Position, charge, reference potential, and reference field values are only read if the corresponding child-elements are present within the `binary` element. The binary particle data is read by all processes in parallel using MPI I/O.

Element	`<binary>...</binary>`
Description	Binary particle data input from a separate file.
Child-elements	`<positions>`, `<charges>`, `<potentials>`, `<field>`

Attributes	**Type**	**Description**
`file`	string	Filename of the input file.
`offset`	int	Offset in bytes within the input file.
`ntotal`	int	Total number of particles to read.

Particle data duplication

The `duplicate` element is used to specify how often the given particles should be duplicated in each dimension. The particle positions can optionally be scaled back to the original size of the system box.

Element	`<duplicate>...</duplicate>`
Description	Explicit duplication of the given input particles (additionally to the duplication specified on the command line).
Child-elements	`<particle>*`, `<binary>*`, `<generate>*`

Attributes	**Type**	**Description**
`times`	int[3]	Number of times the given input particles should be duplicated in each dimension. Value '1 1 1' corresponds to no duplication.
`rescale`	bool	Whether the particle positions should be scaled back to the original system size after the duplication.

Particle data generators

The test program contains several generators that can be used to create particle data for arbitrary large particle configurations. The `generate` element can be used to specify the generation of a set of particles. Particle data generators are executed by all processes in parallel such that each process creates only its locally required particles. The generation of position, charge, reference potential, and reference field values are be specified independently from each other.

Element	`<generate>` ... `</generate>`
Description	Generator for particle data
Child-elements	`<positions>`, `<charges>`, `<potentials>`, `<field>`

Attributes	**Type**	**Description**
`nlocal`	int	Local number of particles to generate on each process.
`ntotal`	int or int[3]	Total number of particles to generate. Three values can be used to specify the number of particles to generate for a grid of particles.

The generation of particle positions consists of the type of input values to use and of a shape in which these input values are transformed to. Both type and shape can be specified as attributes of the `positions` element.

Element	`<positions>` ... `</positions>`
Description	Parameters for generating particle positions.

Attributes	**Type**	**Description**
`type`	string	Use 'random', 'hammersley', or 'halton' to create corresponding sequences of input values. These values are transformed into the specified shape. Use 'grid' to create equally spaced particles. In this case, the three values within the `ntotal` attribute of the enclosing `generate` element are used to specify the number of particles in each dimension of the grid.
`shape`	string	Use the generate sequences of input values to create a particle distribution shaped like a 'box', 'ball', or the surface of a 'sphere'. Furthermore, two way of creating plummer distributions are available ('plummer' and 'plummer_ball'). All distributions are scaled to fit into the given system box.

Particle charges, reference potential, and reference field values can either be set to a single constant value or a list of alternating values.

Element	`<charges>`...`</charges>`, `<potentials>`...`</potentials>`, `<field>`...`</field>`	
Description	Parameters for generating particle charges, reference potentials, and reference field values.	
child-elements	none	

Attributes	**Type**	**Description**
`type`	string	Use 'const' or 'alternate' to set the particle data to a single constant value or to a list of alternating values, respectively.
`value`	float[*]	Constant or alternating value(s) to use.

18.2. Numerical Results

18.3. Generation of Simulation Data

We consider Hammersley- and Halton sequences [13]. These sequences provide pseudo random numbers such that the distance between two particles does not become too small. We use the program *HAMMERSLEY* [1] from John Burkardt to generate these sequences.

These sequences are defined as follows: Let p prime. Each number $k \in \mathbb{N}$ can be uniquely represented as

$$k = a_0 + a_1 p + a_2 p^2 + \ldots + a_r p^r,$$

with $r \in \mathbb{N}$ and $a_i \in \{0, \ldots, p-1\}$, $i = 0, \ldots, r$. The function $\Phi_p(k)$ is given by

$$\Phi_p(k) = \frac{a_0}{p} + \frac{a_1}{p^2} + \frac{a_2}{p^3} + \ldots + \frac{a_r}{p^{r+1}}.$$

Hammersley Sequences Let the dimension d and prime numbers $p_1, p_2, \ldots, p_{d-1}$ with $p_1 < p_2 < \ldots < p_{d-1}$ be given. Furthermore let N the number of particles. The k-th d-dimensional particle of the Hammersley distribution is given by

$$\left(\frac{k}{N}, \Phi_{p_1}(k), \Phi_{p_2}(k), \ldots, \Phi_{p_{d-1}}(k)\right), \; k = 0, \ldots, N-1.$$

Halton Sequences Let in addition a prime number p_d with $p_{d-1} < p_d$ given. The k-th d-dimensional particle of the Halton distribution is given by

$$\left(\Phi_{p_1}(k), \Phi_{p_2}(k), \ldots, \Phi_{p_d}(k)\right), \; k = 0, \ldots, N-1.$$

Since each component of the k-th particle is independent of N, one can produce additional particles.

18.4. Error

In order to demonstrate the efficiency of our methods with a more simple discrete sum we start with the calculation of the potential φ caused by N charged particles with charges q_ℓ given by

$$\varphi(\mathbf{r}_j) = \sum_{\substack{\ell=1 \\ j \neq \ell}}^{N} \frac{q_\ell}{\|\mathbf{r}_j - \mathbf{r}_\ell\|}, \qquad (\mathbf{r}_j \in \mathbb{R}^3).$$

the force

$$\boldsymbol{F}(\boldsymbol{r}_j) := \left(F_0(\boldsymbol{r}_j), F_1(\boldsymbol{r}_j), F_2(\boldsymbol{r}_j)\right)^\top = -q_j \nabla\varphi(\boldsymbol{r}_j)$$

the energy

$$E := \frac{1}{2} \sum_{j=1}^{N} q_j \, \varphi\left(\boldsymbol{r}_j\right) = \frac{1}{2} \sum_{j=1}^{N} q_j \sum_{\substack{\ell=1 \\ \ell \neq j}}^{N} \frac{q_j}{\left\|\boldsymbol{r}_j - \boldsymbol{r}_\ell\right\|_2}$$

and the virial

???

We investigate the absolute energy error

$$\mathcal{E}_{\text{abs}}^{E}(\text{a}, \text{b}) = \left|E_{\text{a}} - E_{\text{b}}\right|,$$

the absolute RMS potential error

$$\mathcal{E}_{\text{abs}}^{\varphi}(\text{a,b}) = \left(\frac{1}{N} \sum_{k=1}^{N} \left|\varphi_{\text{a}}(\mathbf{r}_k) - \varphi_{\text{b}}(\mathbf{r}_k)\right|^2\right)^{1/2},$$

the absolute RMS force error

$$\mathcal{E}_{\text{abs}}^{\boldsymbol{F}}(\text{a,b}) = \left(\frac{1}{N} \sum_{k=1}^{N} \left\|\mathbf{F}_{\text{a}}(\mathbf{r}_k) - \mathbf{F}_{\text{b}}(\mathbf{r}_k)\right\|_2^2\right)^{1/2},$$

the corresponding relative errors

$$\mathcal{E}_{\text{rel}}^{E}(\text{a}, \text{b}) = \frac{\left|E_{\text{a}} - E_{\text{b}}\right|}{\left|E_{\text{a}}\right|},$$

$$\mathcal{E}_{\text{rel}}^{\varphi}(\text{a,b}) = \left(\sum_{k=1}^{N} \left|\varphi_{\text{a}}(\mathbf{r}_k) - \varphi_{\text{b}}(\mathbf{r}_k)\right|^2\right)^{1/2} \left(\sum_{k=1}^{N} \left|\varphi_{\text{a}}(\mathbf{r}_k)\right|^2\right)^{-1/2},$$

$$\mathcal{E}_{\text{rel}}^{\boldsymbol{F}}(\text{a,b}) = \left(\sum_{k=1}^{N} \left\|\mathbf{F}_{\text{a}}(\mathbf{r}_k) - \mathbf{F}_{\text{b}}(\mathbf{r}_k)\right\|_2^2\right)^{1/2} \left(\sum_{k=1}^{N} \left\|\mathbf{F}_{\text{a}}(\mathbf{r}_k)\right\|_2^2\right)^{-1/2},$$

where a and b represent the different techniques for the computation. Furthermore we investigate the sum $\boldsymbol{F}_{\text{S}}(\text{a}) = \sum_{j=1}^{N} \boldsymbol{F}\left(\boldsymbol{r}_j\right)$, which should be zero and compute $F_{\text{S}}(a) := \left\|\boldsymbol{F}_{\text{S}}(\text{a})\right\|_\infty$.

18.5. Distributions and Results

18.5.1. Distribution `hammersley_ball`

We consider Hammersley sequences in the cube $[0,1]^3$ generated by $p_1 = 2$, $p_2 = 3$, and, finally, projected in the ball

$$(x - 0.5)^2 + (y - 0.5)^2 + (z - 0.5)^2 \leq (0.5)^2.$$

We choose the charges $q_\ell \in \{-1;1\}$ random such that

$$\sum_{\ell=1}^{N} q_\ell \in \{-1;0;1\}\,.$$

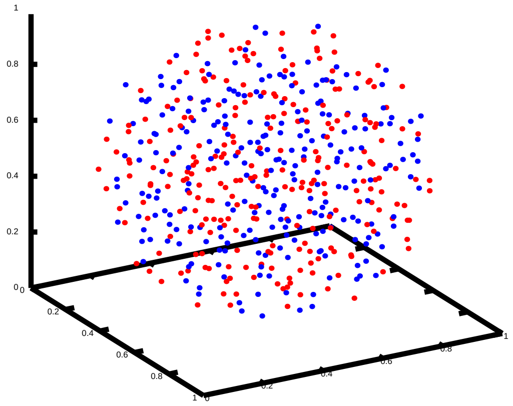

Figure 18.1.: Distribution `hammersley_ball` with 500 randomly charged particles.

We choose the following parameter for our tests.

18.5.2. Distribution `hammersley_ball_neg_charge`

As numerical test we used a ball uniformly filled with charged particles. The total charge of the ball has been kept with $Q = -1$ nC. Thus the particles are assumed to posses the charge $q_\ell = q = -1/N$ nC ($\ell = 1, \ldots, N$), where N denotes the number of particles in the ball. These particles are also regarded as macro-particles representing the distribution of all particles (for instance electrons) in a bunch.

Since a ball uniformly filled with an increasing number of particles of equal charge gets more and more close to a ball with charge $Q = \sum_{\ell=1}^{N} q_\ell$, we compare the results of the summation to the analytically known potential of a homogeneously charged ball given by (normalized by the factor $4\pi\varepsilon_0$)

$$\varphi_{\mathrm{asym}}(\mathbf{r}_j) = \frac{Q}{4\pi\varepsilon_0}\left(\frac{3}{2} - \frac{\|\mathbf{r}_j\|}{2R^2}\right), \qquad \|\mathbf{r}_j\| \le R,$$

where R denotes the radius of the ball.

The numerical experiments documented in Table 18.5 show that we obtain with our fast algorithm the same errors as with the straightforward (slow) summation but with an numerical effort of only $\mathcal{O}(N \log N)$. Hereby the parameters of the P2NFFT are chosen such that the approximation error is less than the simulation error. Finally, we test the algorithm for the computation of the electrostatic field. It is well known that the field of a homogeneously charged ball is given by (normalized by the factor $4\pi\varepsilon_0$)

$$\mathbf{E}_{\mathrm{asym}}(\mathbf{r}_j) = Q\left(\frac{\mathbf{r}_j}{R^3}\right), \qquad \|\mathbf{r}_j\| \le R.$$

	FMM				P2NFFT			
N	???	???	???	???	n	m	p	$\varepsilon_I = \varepsilon_B$
500	???	???	???	???	32,32,32	2	5	3.5/32
5 000	???	???	???	???	32,32,32	2	5	3.5/32
50 000	???	???	???	???	64,64,64	2	5	3.5/64
500 000	???	???	???	???	128,128,128	2	5	3.5/128

Table 18.1.: Parameters for the distribution `hammersley_ball`

N	t_{direct}	t_{FMM}	t_{P2NFFT}	$\mathcal{E}_{rel}^{\varphi}$ (direct, FMM)	$\mathcal{E}_{rel}^{\varphi}$ (direct, P2NFFT)
500	???	???	???	???	???
5 000	???	???	???	???	???
50 000	???	???	???	???	???
500 000	*	???	???	*	*

Table 18.2.: Time and Error for 32 processes

P	t_{FMM}	t_{P2NFFT}	$\mathcal{E}_{rel}^{\varphi}$ (ref, FMM)	$\mathcal{E}_{rel}^{\varphi}$ (ref, P2NFFT)
32	???	???	???	???
128	???	???	???	???
512	???	???	???	???
2048	???	???	???	???
8192	???	???	???	???

Table 18.3.: Strong scaling for $N = 1\,000\,000$ particles and error bound $\mathcal{E}_{\text{rel}}^{\varphi} \leq 10^{-4}$

P	t_{FMM}	t_{P2NFFT}	$\mathcal{E}_{rel}^{\varphi}$ (ref, FMM)	$\mathcal{E}_{rel}^{\varphi}$ (ref, P2NFFT)
32	???	???	???	???
128	???	???	???	???
512	???	???	???	???
2048	???	???	???	???
8192	???	???	???	???

Table 18.4.: Weak scaling for $N = 1\,000\,000$ particles per process and error bound $\mathcal{E}_{\text{rel}}^{\varphi} \leq 10^{-4}$

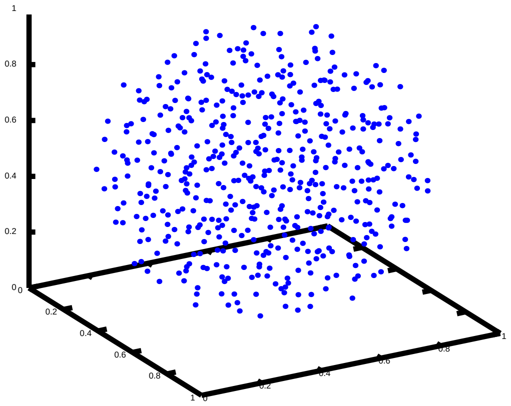

Figure 18.2.: Distribution `hammersley_ball` with 500 negative charged particles.

N	t_{slow}	t_{fast}	$\mathcal{E}^{\varphi}_{rel}$(asym,slow)	$\mathcal{E}^{\varphi}_{rel}$(asym,P2NFFT)	$\mathcal{E}^{\varphi}_{rel}$(slow,P2NFFT)
10 000	???	???	???	???	???
50 000	???	???	???	???	???
100 000	???	???	???	???	???
250 000	*	???	*	???	*
500 000	*	???	*	???	*
1 000 000	*	???	*	???	*

Table 18.5.: Computational time and the error $\mathcal{E}_{\text{rel}}$ for the potential φ, *estimated.

Part III.

Bibliography

19. Bibliography

[1] HAMMERSLEY – The Hammersley Quasirandom Sequence. URL `http://people.scs.fsu.edu/~burkardt/cpp_src/hammersley/hammersley.html`.

[2] pkg-config man page. URL `http://linux.die.net/man/1/pkg-config`. linux manual page for pkg-package tool.

[3] *The Fortran 2003 Handbook - The Complete Syntax, Features and Procedures*. Springer Science+Business Media, 2009.

[4] M. Abramowitz and I. A. Stegun, editors. *Handbook of Mathematical Functions*. National Bureau of Standards, Washington, DC, USA, 1972.

[5] M. Deserno and C. Holm. How to mesh up Ewald sums. I. A theoretical and numerical comparison of various particle mesh routines. *J. Chem. Phys.*, 109:7678 – 7693, 1998.

[6] Daan Frenkel and Berend Smit. *Understanding molecular simulation: From algorithms to applications*. Academic Press, 2002.

[7] Fredrik Hedman and Aatto Laaksonen. Ewald summation based on nonuniform fast Fourier transform. *Chem. Phys. Lett.*, 425:142 – 147, 2006.

[8] Jiri Kolafa and John W. Perram. Cutoff errors in the Ewald summation formulae for point charge systems. *Mol. Simul.*, 9(5):351–368, 1992. URL `http://www.icpf.cas.cz/jiri/papers/ewalderr/default.htm`.

[9] A. C. Maggs and V. Rosseto. Local simulation algorithms for coulombic interactions. *Phys. Rev. Lett.*, 88:196402, 2002.

[10] I. Pasichnyk and B. Dünweg. Coulomb interactions via local dynamics: A molecular-dynamics algorithm. *J. Phys.: Condens. Matter*, 16(38):3999–4020, September 2004.

[11] Michael Pippig and Daniel Potts. Particle simulation based on nonequispaced fast Fourier transforms. In Godehard Sutmann, Paul Gibbon, and Thomas Lippert, editors, *Fast Methods for Long-Range Interactions in Complex Systems*, IAS-Series, pages 131 – 158, Jülich, 2011. Forschungszentrum Jülich. ISBN 9783893367146. URL `http://hdl.handle.net/2128/4441`.

[12] Godehard Sutmann. Molecular dynamics - vision and reality. In Johannes Grotendorst, S. Blügel, and Dominik Marx, editors, *Computational Nanoscience: Do It Yourself!*, volume 31 of *NIC Series*, pages 159 – 194, Jülich, February 2006. Forschungszentrum Jülich, John von Neumann Institute for Computing. ISBN 3-00-017350-1. Lecture Notes.

[13] Tien-Tsin Wong, Wai-Shing Luk, and Pheng-Ann Heng. Sampling with Hammersley and Halton Points. *Journal of graphics tools*, 2(2):9–24, 1997.

Part IV.

Index

List of Tables

1.1. Overview of the features of the different solvers. 13

2.1. List of common `configure` options of the ScaFaCoS library. 17

3.1. Solvers implemented in ScaFaCoS (02/2012) 20
3.2. ScaFaCoS data types for C 21
3.3. ScaFaCoS data kinds for Fortran 22
3.4. ScaFaCoS mathematical macros 23
3.5. ScaFaCoS error values 24
3.6. Parameters for fcs_init 24
3.7. Parameters for fcs_common_set 26

18.1. Parameters for the distribution `hammersley_ball` 107
18.2. Time and Error for 32 processes 107
18.3. Strong scaling for $N = 1\,000\,000$ particles and error bound $\mathcal{E}^{\varphi}_{\mathrm{rel}} \leq 10^{-4}$ . . 107
18.4. Weak scaling for $N = 1\,000\,000$ particles per process and error bound $\mathcal{E}^{\varphi}_{\mathrm{rel}} \leq 10^{-4}$ 107
18.5. Computational time and the error $\mathcal{E}_{\mathrm{rel}}$ for the potential φ, *estimated. . . 108

List of Figures

3.1. Use of the provided header / module of the ScaFaCoS library. 21
3.2. exemplary usage of pkg-config to get the necessary information for compilation 21
3.3. Scheme of system parameters describing the form and position of the simulation box 25
3.4. Example for string truncation for strings used with ScaFaCoS 28
3.5. Example of a parser string setting the first box vector, the near field flag and a method-specific parameter. 29

9.1. Splitting of Coulomb potential into near field (blue) and far field (red). . . 46

13.1. The charge distribution consisting of point charges (left) is split into a smoothened part (right) and the rest (center), see [VMG-1] 71

15.1. Function call: fcs_init 79
15.2. Function call: fcs_common_set 80
15.3. Function call: fcs_tune 80
15.4. Function call: fcs_run 81
15.5. Function call: fcs_destroy 81
15.6. Function call: fcsResult_getReturnCode 82
15.7. Function call: fcsResult_getErrorMessage 82
15.8. Function call: fcsResult_getErrorSource 82
15.9. Function call: fcs_compute_near_field 83
15.10Function call: fcs_compute_near_potential 83
15.11Function call: fcs_compute_near 84
15.12Function call: fcs_method_has_near 84
15.13Function call: fcs_printHandle 85
15.14Function call: fcs_parser 85
15.15Function call: fcs_require_virial 85
15.16Function call: fcs_get_virial 86
15.17Function call: fcs_p3m_compute_near_field 88
15.18Function call: fcs_p3m_compute_near_potential 88
15.19Function call: fcs_p3m_compute_near 89

18.1. Distribution `hammersley_ball` with 500 randomly charged particles. . . . 106
18.2. Distribution `hammersley_ball` with 500 negative charged particles. 108

Index

Build System, 93

C++, 13
C99, 13
Compilation, 15
Configure, 15

data types, **21**
direct, **77**
Direct Summation, **77**

error handling, 27
Ewald, **43**

Fast Multipole Method, **31**
feature matrix, **13**
FFTW, 13
FMM, **31**
Fortran 2003, 13
Functions
    fcs_common_set, **23**, **80**
    fcs_compute_near, **84**
    fcs_compute_near_field, **83**
    fcs_compute_near_potential, **83**
    fcs_destroy, **27**, **81**
    fcs_get_virial, **86**
    fcs_init, **22**, **79**
    fcs_method_has_near, **84**
    fcs_p3m_compute_near, **89**
    fcs_p3m_compute_near_field, **88**
    fcs_p3m_compute_near_potential, **88**
    fcs_parser, **29**, **85**
    fcs_printHandle, **29**, **85**
    fcs_require_virial, **85**
    fcs_run, **27**, **81**
    fcs_tune, **26**, **80**
    fcsResult_getErrorMessage, **82**
    fcsResult_getErrorSource, **82**
    fcsResult_getReturnCode, **82**

Implementation, 97
Installation, 15

Maxwell Equation Molecular Dynamics, **33**
MEMD, **33**
MMM1D, **39**
MMM2D, **41**
MPI, 13
Multigrid, 69, 71

NameExpanded, **69**
near field solver, 28

P^{2}NFFT, **45**
P^{3}M, **61**
P3M, **61**
parameter output, 29
parser, 29
Particle Mesh Ewald, 61
Particle-Particle NFFT, **45**
Particle-Particle Particle-Mesh Ewald, **61**
PEPC, **63**
pp3mg, **69**
Pretty Efficient Parallel Coulomb Solver, **63**

requirements, **12**

Solver
    data types, **21**
    direct, **77**

  - Direct Summation, **77**
  - Ewald, **43**
  - Fast Multipole Method, **31**
  - FMM, **31**
  - Maxwell Equation Molecular Dynamics, **33**
  - MEMD, **33**
  - MMM1D, **39**
  - MMM2D, **41**
  - NameExpanded, **69**
  - P^{2}NFFT, **45**
  - P^{3}M, **61**
  - P3M, **61**
  - Particle-Particle NFFT, **45**
  - Particle-Particle Particle-Mesh Ewald, **61**
  - PEPC, **63**
  - pp3mg, **69**
  - Pretty Efficient Parallel Coulomb Solver, **63**
  - Versatile Multigrid, **71**
  - vmg, **71**

Versatile Multigrid, **71**
vmg, **71**