MRCPP

Release 1.2.0-alpha

Luca Frediani, Stig Rune Jensen, Peter Wind, Magnar Bjorgve, Ro

May 24, 2023

INSTALLATION

The MultiResolution Computation Program Package (MRCPP) is a general purpose numerical mathematics library based on multiresolution analysis and the multiwavelet basis which provide low-scaling algorithms as well as rigorous error control in numerical computations.

The code is being developed at the Hylleraas Centre for Quantum Molecular Sciences at UiT - The Arctic University of Norway.

The code can be found on GitHub.

CHAPTER

ONE

OBTAINING THE CODE

The latest version of MRCPP is available on GitHub:

```
$ git clone git@github.com:MRChemSoft/mrcpp.git
```

CHAPTER

TWO

BUILDING THE CODE

2.1 Prerequisites

- g++-5.4 or later (`std=c++14`)
- CMake version 3.11 or higher.
- Eigen version 3.3 or higher.
- BLAS (optional)

2.2 Configuration

The configuration and build process is managed through CMake, and a `setup` script is provided for the configuration step. MRCPP's only dependency is Eigen3, which will be downloaded at configure time unless it is already available on the system. If you have a local version *not* under the system path, you can point to it before running `setup`:

```
$ export EIGEN3_ROOT=/path/to/eigen3
$ ./setup [options] [<builddir>]
```

The setup script will create a directory called <*builddir*> (default `build`) and run CMake. There are several options available for the setup, and the most important are:

--cxx=<CXX>
: C++ compiler [default: g++]

--omp
: Enable OpenMP parallelization [default: False]

--mpi
: Enable MPI parallelization [default: False]

--enable-tests
: Enable tests [default: True]

--enable-examples
: Enable tests [default: False]

--type=<TYPE>
: Set the CMake build type (debug, release, relwithdebinfo, minsizerel) [default: release]

--prefix=<PATH>
: Set the install path for make install

-h --help
List all options

2.3 Compilation

After successful configuration, the code is compiled using the `make` command in the <*builddir*> directory:

```
$ cd <builddir>
$ make
```

CHAPTER

THREE

RUNNING TESTS

A set of tests is provided with the code to verify that the code compiled properly. To compile the test suite, add the `--enable-tests` option to setup, then run the tests with `ctest`:

```
$ ./setup --enable-tests build
$ cd build
$ make
$ ctest
```

CHAPTER

FOUR

RUNNING EXAMPLES

In addition to the test suite, the code comes with a number of small code snippets that demonstrate the features and the API of the library. These are located in the `examples` directory. To compile the example codes, add the `enable-examples` option to setup, and the example executables can be found under `<build-dir>/bin/`. E.g. to compile and run the MW projection example:

```
$ ./setup --enable-examples build-serial
$ cd build-serial
$ make
$ bin/projection
```

The shared memory parallelization (OpenMP) is controlled by the environment variable `OMP_NUM_THREADS` (make sure you have compiled with the `--omp` option to setup). E.g. to compile and run the Poisson solver example using 10 CPU cores:

```
$ ./setup --enable-examples --omp build-omp
$ cd build-omp
$ make
$ OMP_NUM_THREADS=10 bin/poisson
```

To run in MPI parallel, use the `mpirun` (or equivalent) command (make sure you have compiled with the `--mpi` option to setup, and used MPI compatible compilers, e.g. `--cxx=mpicxx`). Only examples with an *mpi* prefix will be affected by running in MPI:

```
$ ./setup --cxx=mpicxx --enable-examples --mpi build-mpi
$ cd build-mpi
$ make
$ mpirun -np 4 bin/mpi_send_tree
```

To run in hybrid OpenMP/MPI parallel, simply combine the two above:

```
$ ./setup --cxx=mpicxx --enable-examples --omp --mpi build-hybrid
$ cd build-hybrid
$ make
$ export OMP_NUM_THREADS=5
$ mpirun -np 4 bin/mpi_send_tree
```

Note that the core of MRCPP is *only* OpenMP parallelized. All MPI data or work distribution must be done manually in the application program, using the tools provided by MRCPP (see the Parallel section of the API).

CHAPTER

FIVE

PILOT CODE

Finally, MRCPP comes with a personal sandbox where you can experiment and test new ideas, without messing around in the git repository. In the `pilot/` directory you will find a skeleton code called `mrcpp.cpp.sample`. To trigger a build, re-name (copy) this file to `mrcpp.cpp`:

```
$ cd pilot
$ cp mrcpp.cpp.sample mrcpp.cpp
```

Now a corresponding executable will be build in `<builddir>/bin/mrcpp-pilot/`. Feel free to do whatever you like in your own pilot code, but please don't add this file to git. Also, please don't commit any changes to the existing examples (unless you know what you're doing).

CHAPTER

SIX

MRCPP AS A DEPENDENCY

Building MRCPP provides CMake configuration files exporting the libraries and headers as targets to be consumed by third-party projects also using CMake:

```
$ ./setup --prefix=$HOME/Software/mrcpp
$ cd build
$ make
$ ctest
$ make install
```

Now libraries, headers and CMake configuration files can be found under the given prefix:

```
mrcpp/
├── include/
│   └── MRCPP/
├── lib64/
│   ├── libmrcpp.a
│   ├── libmrcpp.so -> libmrcpp.so.1*
│   └── libmrcpp.so.1*
└── share/
    └── cmake/
```

As an example, the `pilot` sample can be built with the following `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.11 FATAL_ERROR)

project(UseMRCPP LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include(GNUInstallDirs)

set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_LIBDIR})
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/${CMAKE_INSTALL_BINDIR})

find_package(MRCPP CONFIG REQUIRED)
get_property(_loc TARGET MRCPP::mrcpp PROPERTY LOCATION)
message(STATUS "Found MRCPP: ${_loc} (found version ${MRCPP_VERSION})")

```

(continues on next page)

(continued from previous page)

```
add_executable(mrcpp mrcpp.cpp)
target_link_libraries(mrcpp
  PUBLIC
  MRCPP::mrcpp
  )

set_target_properties(mrcpp
  PROPERTIES
    MACOSX_RPATH ON
    SKIP_BUILD_RPATH OFF
    BUILD_WITH_INSTALL_RPATH OFF
    INSTALL_RPATH "$ORIGIN/../${CMAKE_INSTALL_LIBDIR}"
    INSTALL_RPATH_USE_LINK_PATH ON
  )
```

This will set up the include paths and library paths correctly. During configuration you will have to specify *where* the CMake configuration file for MRCPP is located:

```
$ cmake -H. -Bbuild -DMRCPP_DIR=$HOME/Software/share/cmake/MRCPP
```

CHAPTER

SEVEN

INTRODUCTION

The main features of MRCPP are the numerical multiwavelet (MW) representations of functions and operators. Two integral convolution operators are implemented (the Poisson and Helmholtz operators), as well as the partial derivative and arithmetic operators. In addition to the numerical representations there are a limited number of analytic functions that are usually used as starting point for the numerical computations. Also, MRCPP provides three convenience classes (Timer, Printer and Plotter) that can be made available to the application program.

The API consists of seven include files which will be discussed in more detail below:

```
MRCPP/
├── MWFunctions
├── MWOperators
├── Gaussians
├── Parallel
├── Printer
├── Plotter
└── Timer
```

MRCPP/MWFunctions
: Provides features for representation and manipulation of real-valued scalar functions in a MW basis, including projection of analytic function, numerical integration and scalar products, as well as arithmetic operations and function mappings.

MRCPP/MWOperators
: Provides features for representation and application of MW operators. Currently there are three operators available: Poisson, Helmholtz and Cartesian derivative.

MRCPP/Gaussians
: Provides some simple features for analytical Gaussian functions, useful e.g. to generate initial guesses for MW computations.

MRCPP/Parallel
: Provides some simple MPI features for MRCPP, in particular the possibility to send complete MW function representations between MPI processes.

MRCPP/Printer
: Provides simple (parallel safe) printing options. All MRCPP internal printing is done with this class, and the printer must be initialized in order to get any printed output, otherwise MRCPP will run silently.

MRCPP/Plotter
: Provides options to generate data files for plotting of MW function representations. These include line plots, surface plots and cube plots, as well as grid visualization using geomview.

MRCPP/Timer
: Provides an accurate timer for the wall clock in parallel computations.

7.1 Analytic functions

The general way of defining an analytic function in MRCPP is to use lambdas (or std::function), which provide lightweight functions that can be used on the fly. However, some analytic functions, like Gaussians, are of special importance and have been explicitly implemented with additional functionality (see Gaussian chapter).

In order to be accepted by the MW projector (see MWFunctions chapter), the lambda must have the following signature:

```
auto f = [] (const mrcpp::Coord<D> &r) -> double;
```

e.i. it must take a D-dimensional Cartesian coordinate (`mrcpp::Coord<D>` is simply an alias for `std::array<double, D>`), and return a `double`. For instance, the electrostatic potential from a point nuclear charge Z (in atomic units) is

$$f(r) = \frac{Z}{r}$$

which can be written as the lambda function

```
auto Z = 1.0; // Hydrogen nuclear charge
auto f = [Z] (const mrcpp::Coord<3> &r) -> double {
    auto R = std::sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
    return Z/R;
};
```

Note that the function signature must be *exactly* as given above, which means that any additional arguments (such as Z in this case) must be given in the capture list (square brackets), see e.g. cppreference.com for more details on lambda functions and how to use the capture list.

CHAPTER

EIGHT

MWFUNCTIONS

Everything that is discussed in the following chapter is available to the application program by including:

```
#include "MRCPP/MWFunctions"
```

Multiwavelet (MW) representations of real-valued scalar functions are in MRCPP called `FunctionTrees`. These are in principle available in any dimension using the template parameter D (in practice D=1,2,3). There are several different ways of constructing MW functions (computing the expansion coefficients in the MW basis):

- Projection of analytic function
- Arithmetic operations
- Application of MW operator

The first two will be discribed in this chapter, while the last one regarding operators will be the topic of the next chapter.

The interface for constructing MW representations in MRCPP has a dual focus: on the one hand we want a simple, intuitive way of producing adaptive numerical approximations with guaranteed precision that does not require detailed knowledge of the internals of the MW code and with a minimal number of parameters that have to be set. On the other hand we want the possibility for more detailed control of the construction and refinement of the numerical grid where such control is possible and even necessary. In the latter case it is important to be able to reuse the existing grids in e.g. iterative algorithms without excessive allocation/deallocation of memory.

8.1 MultiResolution Analysis (MRA)

template<int **D**>

class **MultiResolutionAnalysis**

Class collecting computational domain and MW basis.

In order to combine different functions and operators in mathematical operations, they need to be compatible. That is, they must be defined on the same computational domain and constructed using the same polynomial basis (order and type). This information constitutes an MRA, which needs to be defined and passed as argument to all function and operator constructors, and only functions and operators with compatible MRAs can be combined in subsequent calculations.

Public Functions

MultiResolutionAnalysis(const *BoundingBox*<*D*> &bb, const ScalingBasis &sb, int depth = MaxDepth)

Parameters

- **bb** – **[in]** Computational domain
- **sb** – **[in]** Polynomial basis
- **depth** – **[in]** Maximum allowed resolution depth, relative to root scale

Returns
New *MultiResolutionAnalysis* object

template<int **D**>

class **BoundingBox**

Class defining the computational domain.

The computational domain is made up of a collection of D-dimensional boxes on a particular length scale n. The size of each box is then $[2^{-n}]^D$, i.e. higher scale means smaller boxes, and the scale may be negative. The number of boxes can be different in each dimension $[n_x, n_y, \dots]$, but they must all be on the same scale (size). Box translations relative to the world origin *must* be an integer multiple of the given scale size 2^{-n}.

Subclassed by mrcpp::NodeBox< D >

Public Functions

explicit **BoundingBox**(std::array<int, 2> box)

Creates a box with appropriate root scale and scaling factor to fit the given bounds, which applies to *all* dimensions. Root scale is chosen such that the scaling factor becomes 1 < sfac < 2.

Limitations: Box must be *either* [0,L] *or* [-L,L], with L a positive integer.

Parameters
box – **[in]** [lower, upper] bound in all dimensions

Returns
New *BoundingBox* object

explicit **BoundingBox**(int n = 0, const std::array<int, *D*> &l = {}, const std::array<int, *D*> &nb = {}, const std::array<double, *D*> &sf = {}, bool pbc = false)

Parameters

- **n** – **[in]** Length scale, default 0
- **l** – **[in]** Corner translation, default [0, 0, ...]
- **nb** – **[in]** Number of boxes, default [1, 1, ...]
- **sf** – **[in]** Scaling factor, default [1.0, 1.0, ...]

Returns
New *BoundingBox* object

class **LegendreBasis** : public mrcpp::ScalingBasis

Legendre scaling functions as defined by Alpert, SIAM J Math Anal 24 (1), 246 (1993).

Public Functions

inline **LegendreBasis**(int k)

Parameters
k – **[in]** Polynomial order of basis, `1 < k < 40`

Returns
New *LegendreBasis* object

class **InterpolatingBasis** : public mrcpp::ScalingBasis

Interpolating scaling functions as defined by Alpert etal, J Comp Phys 182, 149-190 (2002).

Public Functions

inline **InterpolatingBasis**(int k)

Parameters
k – **[in]** Polynomial order of basis, `1 < k < 40`

Returns
New *InterpolatingBasis* object

8.2 FunctionTree

template<int **D**>

class **FunctionTree** : public mrcpp::MWTree<*D*>, public mrcpp::RepresentableFunction<*D*>

Function representation in MW basis.

Constructing a full grown *FunctionTree* involves a number of steps, including setting up a memory allocator, constructing root nodes according to the given MRA, building an adaptive tree structure and computing MW coefficients. The *FunctionTree* constructor does only half of these steps: It takes an MRA argument, which defines the computational domain and scaling basis (these are fixed parameters that cannot be changed after construction). The tree is initialized with a memory allocator and a set of root nodes, but it does not compute any coefficients and the function is initially *undefined*. An undefined *FunctionTree* will have a well defined tree structure (at the very least the root nodes of the given MRA, but possibly with additional refinement) and its MW coefficient will be allocated but uninitialized, and its square norm will be negative (minus one).

Public Functions

FunctionTree(const *MultiResolutionAnalysis*<*D*> &mra, *SharedMemory* *sh_mem = nullptr, const std::string &name = "nn")

Constructs an uninitialized tree, containing only empty root nodes. If a shared memory pointer is provided the tree will be allocated in this shared memory window, otherwise it will be local to each MPI process.

Parameters

- **mra** – **[in]** Which MRA the function is defined
- **sh_mem** – **[in]** Pointer to MPI shared memory block

Returns
New *FunctionTree* object

8.2.1 Creating defined FunctionTrees

The following functions will *define* MW coefficients where there are none, and thus *require* that the output `FunctionTree` is in an *undefined* state. All functions marked with 'adaptive grid' will use the same building algorithm:

1. Start with an initial guess for the grid
2. Compute the MW coefficients for the output function on the current grid
3. Refine the grid where necessary based on the local wavelet norm
4. Iterate points 2 and 3 until the grid is converged

With a *negative* precision argument, the grid will be *fixed*, e.i. it will not be refined beyond the initial grid. There is also an argument to limit the number of *extra* refinement levels beyond the initial grid, in which the adaptive refinement will stop, even if the local precision requirement is not met.

void `mrcpp::MWTree::setZero()`

Set the MW coefficients to zero, fixed grid.

Keeps the node structure of the tree, even though the zero function is representable at depth zero. Use cropTree to remove unnecessary nodes.

template<int **D**>
void `mrcpp::project`(double prec, *FunctionTree*<*D*> &out, RepresentableFunction<*D*> &inp, int maxIter, bool absPrec)

Project an analytic function onto the MW basis, adaptive grid.

The output function will be computed using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **inp** – **[in]** Input function
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision

template<int **D**>

void mrcpp::**copy_func**(*FunctionTree*<*D*> &out, *FunctionTree*<*D*> &inp)

Copy function from one tree onto the grid of another tree, fixed grid.

The output function will be computed using the general algorithm:

- Loop through current leaf nodes of the output tree
- Copy MW coefs from the corresponding input node

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called and will overwrite any existing coefs.

Parameters

- **out** – **[out]** Output function
- **inp** – **[in]** Input function

template<int **D**>
void mrcpp::**add**(double prec, *FunctionTree*<*D*> &out, FunctionTreeVector<*D*> &inp, int maxIter, bool absPrec)

Addition of several MW function representations, adaptive grid.

The output function will be computed as the sum of all input functions in the vector (including their numerical coefficients), using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **inp** – **[in]** Vector of input function
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision

template<int **D**>

void mrcpp::add(double prec, *FunctionTree*<*D*> &out, double a, *FunctionTree*<*D*> &inp_a, double b, *FunctionTree*<*D*> &inp_b, int maxIter, bool absPrec)

Addition of two MW function representations, adaptive grid.

The output function will be computed as the sum of the two input functions (including the numerical coefficient), using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **a** – **[in]** Numerical coefficient of function a
- **inp_a** – **[in]** Input function a
- **b** – **[in]** Numerical coefficient of function b
- **inp_b** – **[in]** Input function b
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision

template<int **D**>
void mrcpp::multiply(double prec, *FunctionTree*<*D*> &out, FunctionTreeVector<*D*> &inp, int maxIter, bool absPrec, bool useMaxNorms)

Multiplication of several MW function representations, adaptive grid.

The output function will be computed as the product of all input functions in the vector (including their numerical coefficients), using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **inp** – **[in]** Vector of input function
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision
- **useMaxNorms** – **[in]** Build output tree based on norm estimates from input

template<int **D**>
void `mrcpp::multiply`(double prec, *FunctionTree*<*D*> &out, double c, *FunctionTree*<*D*> &inp_a, *FunctionTree*<*D*> &inp_b, int maxIter, bool absPrec, bool useMaxNorms)

Multiplication of two MW function representations, adaptive grid.

The output function will be computed as the product of the two input functions (including the numerical coefficient), using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **c** – **[in]** Numerical coefficient
- **inp_a** – **[in]** Input function a
- **inp_b** – **[in]** Input function b
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision
- **useMaxNorms** – **[in]** Build output tree based on norm estimates from input

template<int **D**>

void mrcpp::square(double prec, *FunctionTree*<*D*> &out, *FunctionTree*<*D*> &inp, int maxIter, bool absPrec)

Out-of-place square of MW function representations, adaptive grid.

The output function will be computed as the square of the input function, using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **inp** – **[in]** Input function to square
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision

template<int **D**>
void mrcpp::power(double prec, *FunctionTree*<*D*> &out, *FunctionTree*<*D*> &inp, double p, int maxIter, bool absPrec)

Out-of-place power of MW function representations, adaptive grid.

The output function will be computed as the input function raised to the given power, using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **inp** – **[in]** Input function to square

- **p** – **[in]** Numerical power
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision

template<int D>
void mrcpp::**dot**(double prec, *FunctionTree*<*D*> &out, FunctionTreeVector<*D*> &inp_a, FunctionTreeVector<*D*> &inp_b, int maxIter, bool absPrec)

Dot product of two MW function vectors, adaptive grid.

The output function will be computed as the dot product of the two input vectors (including their numerical coefficients). The precision parameter is used only in the multiplication part, the final addition will be on the fixed union grid of the components.

Note: The length of the input vectors must be the same.

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **inp_a** – **[in]** Input function vector
- **inp_b** – **[in]** Input function vector
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision

template<int D>
void mrcpp::**map**(double prec, *FunctionTree*<*D*> &out, *FunctionTree*<*D*> &inp, FMap fmap, int maxIter, bool absPrec)

map a MW function onto another representations, adaptive grid

The output function tree will be computed by mapping the input tree values through the fmap function, using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

No assumption is made for how the mapping function looks. It is left to the end-user to guarantee that the mapping function does not lead to numerically unstable/inaccurate situations (e.g. divide by zero, overflow, etc…)

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **inp** – **[in]** Input function
- **fmap** – **[in]** mapping function
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree
- **absPrec** – **[in]** Build output tree based on absolute precision

8.2.2 Creating undefined FunctionTrees

The grid of a `FunctionTree` can also be constructed *without* computing any MW coefficients:

template<int **D**>
void `mrcpp::build_grid`(*FunctionTree*<*D*> &out, const RepresentableFunction<*D*> &inp, int maxIter)

Build empty grid based on info from analytic function.

The grid of the output function will be EXTENDED using the general algorithm:

- Loop through current leaf nodes of the output tree
- Refine node based on custom split check from the function
- Repeat until convergence or `maxIter` is reached
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called. It requires that the functions `isVisibleAtScale()` and `isZeroOnInterval()` is implemented in the particular `RepresentableFunction`.

Parameters

- **out** – **[out]** Output tree to be built
- **inp** – **[in]** Input function
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree

template<int **D**>
void `mrcpp::build_grid`(*FunctionTree*<*D*> &out, const *GaussExp*<*D*> &inp, int maxIter)

Build empty grid based on info from Gaussian expansion.

The grid of the output function will be EXTENDED using the general algorithm:

- Loop through current leaf nodes of the output tree
- Refine node based on custom split check from the function
- Repeat until convergence or `maxIter` is reached
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called. It will loop through the Gaussians in the expansion and extend the grid based on the position and exponent of each term. Higher exponent means finer resolution.

Parameters

- **out** – **[out]** Output tree to be built
- **inp** – **[in]** Input Gaussian expansion
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree

template<int **D**>
void **mrcpp::build_grid**(*FunctionTree*<*D*> &out, *FunctionTree*<*D*> &inp, int maxIter)

Build empty grid based on another MW function representation.

The grid of the output function will be EXTENDED with all existing nodes in corresponding input function, using the general algorithm:

- Loop through current leaf nodes of the output tree
- Refine node if the corresponding node in the input has children
- Repeat until all input nodes are covered or `maxIter` is reached
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called. This means that all nodes on the input tree will also be in the final output tree (unless `maxIter` is reached, but NOT vice versa.

Parameters

- **out** – **[out]** Output tree to be built
- **inp** – **[in]** Input tree
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree

template<int **D**>
void **mrcpp::build_grid**(*FunctionTree*<*D*> &out, FunctionTreeVector<*D*> &inp, int maxIter)

Build empty grid based on several MW function representation.

The grid of the output function will be EXTENDED with all existing nodes in all corresponding input functions, using the general algorithm:

- Loop through current leaf nodes of the output tree
- Refine node if the corresponding node in one of the inputs has children
- Repeat until all input nodes are covered or `maxIter` is reached
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called. This means that the final output grid will contain (at least) the union of the nodes of all input trees (unless `maxIter` is reached).

Parameters

- **out** – **[out]** Output tree to be built
- **inp** – **[in]** Input tree vector
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree

template<int **D**>
void `mrcpp::`**copy_grid**(*FunctionTree*<*D*> &out, *FunctionTree*<*D*> &inp)

Build empty grid that is identical to another MW grid.

Note: The difference from the corresponding `build_grid` function is that this will first clear the grid of the `out` function, while `build_grid` will *extend* the existing grid.

Parameters

- **out** – **[out]** Output tree to be built
- **inp** – **[in]** Input tree

template<int **D**>
void `mrcpp::`**clear_grid**(*FunctionTree*<*D*> &out)

Clear the MW coefficients of a function representation.

Note: This will only clear the MW coefs in the existing nodes, it will not change the grid of the function. Use *FunctionTree::clear()* to remove all grid refinement as well.

Parameters

out – **[inout]** Output function to be cleared

void `mrcpp::`*FunctionTree*`::`**clear**()

Remove all nodes in the tree.

Leaves the tree inn the same state as after construction, i.e. undefined function containing only root nodes without coefficients. The assigned memory (nodeChunks in NodeAllocator) is NOT released, but is immediately available to the new function.

8.2.3 Changing FunctionTrees

There are also a number of in-place operations that *change* the MW coefficients of a given defined `FunctionTree`. All changing operations *require* that the `FunctionTree` is in a *defined* state.

void `mrcpp::FunctionTree::rescale(double c)`

In-place multiplication by a scalar, fixed grid.

The leaf node point values of the function will be in-place multiplied by the given coefficient, no grid refinement.

Parameters

c – **[in]** Scalar coefficient

void `mrcpp::FunctionTree::normalize()`

In-place rescaling by a function norm $||f||^{-1}$, fixed grid.

void `mrcpp::FunctionTree::add(double c, FunctionTree<D> &inp)`

In-place addition with MW function representations, fixed grid.

The input function will be added in-place on the current grid of the function, i.e. no further grid refinement.

Parameters

- **c** – **[in]** Numerical coefficient of input function
- **inp** – **[in]** Input function to add

void `mrcpp::FunctionTree::multiply(double c, FunctionTree<D> &inp)`

In-place multiplication with MW function representations, fixed grid.

The input function will be multiplied in-place on the current grid of the function, i.e. no further grid refinement.

Parameters

- **c** – **[in]** Numerical coefficient of input function
- **inp** – **[in]** Input function to multiply

void `mrcpp::FunctionTree::square()`

In-place square of MW function representations, fixed grid.

The leaf node point values of the function will be in-place squared, no grid refinement.

void `mrcpp::FunctionTree::power(double p)`

In-place power of MW function representations, fixed grid.

The leaf node point values of the function will be in-place raised to the given power, no grid refinement.

Parameters

p – **[in]** Numerical power

void `mrcpp::FunctionTree::map(FMap fmap)`

In-place mapping with a predefined function f(x), fixed grid.

The input function will be mapped in-place on the current grid of the function, i.e. no further grid refinement.

Parameters
fmap – **[in]** mapping function

int mrcpp::*FunctionTree*::**crop**(double prec, double splitFac = 1.0, bool absPrec = true)

Reduce the precision of the tree by deleting nodes.

This will run the tree building algorithm in "reverse", starting from the leaf nodes, and perform split checks on each node based on the given precision and the local wavelet norm.

Note: The splitting factor appears in the threshold for the wavelet norm as $||w|| < 2^{-sn/2}||f||\epsilon$. In principal, s should be equal to the dimension; in practice, it is set to s=1.

Parameters

- **prec** – New precision criterion
- **splitFac** – Splitting factor: 1, 2 or 3
- **absPrec** – Use absolute precision

template<int **D**>
int mrcpp::**refine_grid**(*FunctionTree*<*D*> &out, int scales)

Refine the grid of a MW function representation.

This will split ALL leaf nodes in the tree the given number of times, then it will compute scaling coefs of the new nodes, thus leaving the function representation unchanged, but on a larger grid.

Parameters

- **out** – **[inout]** Output tree to be refined
- **scales** – **[in]** Number of refinement levels

Returns
The number of nodes that were split

template<int **D**>
int mrcpp::**refine_grid**(*FunctionTree*<*D*> &out, double prec, bool absPrec)

Refine the grid of a MW function representation.

This will first perform a split check on the existing leaf nodes in the tree based on the provided precision parameter, then it will compute scaling coefs of the new nodes, thus leaving the function representation unchanged, but (possibly) on a larger grid.

Parameters

- **out** – **[inout]** Output tree to be refined
- **prec** – **[in]** Precision for initial split check
- **absPrec** – **[in]** Build output tree based on absolute precision

Returns
The number of nodes that were split

template<int **D**>

int mrcpp::**refine_grid**(*FunctionTree*<*D*> &out, *FunctionTree*<*D*> &inp)

Refine the grid of a MW function representation.

This will first perform a split check on the existing leaf nodes in the output tree based on the structure of the input tree (same as build_grid), then it will compute scaling coefs of the new nodes, thus leaving the function representation unchanged, but on a larger grid.

Parameters

- **out** – **[inout]** Output tree to be refined
- **inp** – **[in]** Input tree that defines the new grid

Returns

The number of nodes that were split

8.2.4 File I/O

void mrcpp::*FunctionTree*::**saveTree**(const std::string &file)

Write the tree structure to disk, for later use.

Parameters

file – **[in]** File name, will get ".tree" extension

void mrcpp::*FunctionTree*::**loadTree**(const std::string &file)

Read a previously stored tree structure from disk.

Note: This tree must have the exact same MRA the one that was saved

Parameters

file – **[in]** File name, will get ".tree" extension

8.2.5 Extracting data

Given a `FunctionTree` that is a *well defined* function representation, the following data can be extracted:

double mrcpp::*FunctionTree*::**integrate**() const

Returns

Integral of the function over the entire computational domain

virtual double mrcpp::*FunctionTree*::**evalf**(const Coord<D> &r) const override

Note: This will only evaluate the *scaling* part of the leaf nodes in the tree, which means that the function values will not be fully accurate. This is done to allow a fast and const function evaluation that can be done in OMP parallel. If you want to include also the *final* wavelet part you can call the corresponding evalf_precise function, *or* you can manually extend the MW grid by one level before evaluating, using `mrcpp::refine_grid(tree, 1)`

Parameters

r – **[in]** Cartesian coordinate

Returns
Function value in a point, out of bounds returns zero

inline double `mrcpp::MWTree::getSquareNorm()` const

Returns
Squared L2 norm of the function

inline int `mrcpp::MWTree::getNNodes()` const

int `mrcpp::MWTree::getSizeNodes()` const

Returns
Size of all MW coefs in the tree, in kB

template<int **D**>
double `mrcpp::dot`(*FunctionTree*<*D*> &bra, *FunctionTree*<*D*> &ket)

The dot product is computed with the trees in compressed form, i.e. scaling coefs only on root nodes, wavelet coefs on all nodes. Since wavelet functions are orthonormal through ALL scales and the root scaling functions are orthonormal to all finer level wavelet functions, this becomes a rather efficient procedure as you only need to compute the dot product where the grids overlap.

Parameters

- **bra** – **[in]** Bra side input function
- **ket** – **[in]** Ket side input function

Returns
Dot product <bra|ket> of two MW function representations

8.3 FunctionTreeVector

The `FunctionTreeVector` is simply an alias for a `std::vector` of `std::tuple` containing a numerical coefficient and a `FunctionTree` pointer.

template<int **D**>
void `mrcpp::clear`(FunctionTreeVector<*D*> &fs, bool dealloc = false)

Remove all entries in the vector.

Parameters

- **fs** – **[in]** Vector to clear
- **dealloc** – **[in]** Option to free *FunctionTree* pointer before clearing

template<int **D**>
double `mrcpp::get_coef`(const FunctionTreeVector<*D*> &fs, int i)

Parameters

- **fs** – **[in]** Vector to fetch from
- **i** – **[in]** Position in vector

Returns
Numerical coefficient at given position in vector

template<int **D**>

FunctionTree<*D*> &**mrcpp::get_func**(FunctionTreeVector<*D*> &fs, int i)

Parameters

- **fs** – **[in]** Vector to fetch from
- **i** – **[in]** Position in vector

Returns

FunctionTree at given position in vector

template<int **D**>
int **mrcpp::get_n_nodes**(const FunctionTreeVector<*D*> &fs)

Parameters

fs – **[in]** Vector to fetch from

Returns

Total number of nodes of all trees in the vector

template<int **D**>
int **mrcpp::get_size_nodes**(const FunctionTreeVector<*D*> &fs)

Parameters

fs – **[in]** Vector to fetch from

Returns

Total size of all trees in the vector, in kB

8.4 Examples

8.4.1 Constructing an MRA

An MRA is defined in two steps, first the computational domain is given by a `BoundingBox` (D is the dimension), e.g. for a total domain of $[-32, 32]^3$ in three dimensions (eight root boxes of size $[16]^3$ each):

```
int n = -4;                                     // Root scale defines box size 2^{-n}
std::array<int, 3> l{-1, -1, -1};               // Translation of first box [l_x,l_y,
↪l_z]
std::array<int, 3> nb{2, 2, 2};                 // Number of boxes [n_x,n_y,n_z]
mrcpp::BoundingBox<3> world(n, l, nb);
```

which is combined with a `ScalingBasis` to give an MRA, e.g. interpolating scaling functions of order $k = 9$:

```
int N = 20;                                     // Maximum refinement 2^{-(n+N)}
int k = 9;                                      // Polynomial order
mrcpp::InterpolatingBasis basis(k);             // Legendre or Interpolating basis
mrcpp::MultiResolutionAnalysis<D> MRA(world, basis, N);
```

Two types of `ScalingBasis` are supported (`LegendreBasis` and `InterpolatingBasis`), and they are both available at orders $k = 1, 2, \ldots, 40$ (note that some operators are constructed using intermediates of order $2k$, so in that case the maximum order available is $k = 20$).

8.4.2 Working withFunctionTreeVectors

Elements can be appended to the vector using the `std::make_tuple`, elements are obtained with the `get_func` and `get_coef` functions:

```
mrcpp::FunctionTreeVector<D> tree_vec;              // Initialize empty vector
tree_vec.push_back(std::make_tuple(2.0, &tree_a)); // Push back pointer to FunctionTree
auto coef = mrcpp::get_coef(tree_vec, 0);          // Get coefficient of first entry
auto &tree = mrcpp::get_func(tree_vec, 0);         // Get function of first entry
mrcpp::clear(tree_vec, false);                     // Bool argument for tree destruction
```

8.4.3 Building empty grids

Sometimes it is useful to construct an empty grid based on some available information of the function that is about to be represented. This can be e.g. that you want to copy the grid of an existing `FunctionTree` or that an analytic function has more or less known grid requirements (like Gaussians). Sometimes it is even necessary to force the grid refinement beyond the coarsest scales in order for the adaptive refining algorithm to detect a wavelet "signal" that allows it to do its job properly (this happens for narrow Gaussians where none of the initial quadrature points hits a function value significantly different from zero).

The simplest way to build an empty grid is to copy the grid from an existing tree (assume that `f_tree` has been properly built so that it contains more than just root nodes)

```
mrcpp::FunctionTree<D> f_tree(MRA);   // Input tree
mrcpp::FunctionTree<D> g_tree(MRA);   // Output tree

mrcpp::project(prec, f_tree, f_func); // Build adaptive grid for f_tree
mrcpp::copy_grid(g_tree, f_tree);     // Copy grid from f_tree to g_tree
```

Passing an analytic function as argument to the generator will build a grid based on some predefined information of the function (if there is any, otherwise it will do nothing)

```
mrcpp::RepresentableFunction<D> func; // Analytic function
mrcpp::FunctionTree<D> tree(MRA);     // Output tree
mrcpp::build_grid(tree, func);        // Build grid based on f_func
```

The lambda analytic functions do *not* provide such information, this must be explicitly implemented as a `RepresentableFunction` sub-class (see MRCPP programmer's guide for details).

Actually, the effect of the `build_grid` is to *extend* the existing grid with any missing nodes relative to the input. There is also a version of `build_grid` taking a `FunctionTree` argument. Its effect is very similar to the `copy_grid` above, with the only difference that now the output grid is *extended* with the missing nodes (e.i. the nodes that are already there are *not* removed first). This means that we can build the union of two grids by successive applications of `build_grid`

```
mrcpp::FunctionTree<D> f_tree(MRA);   // Construct empty grid of root nodes
mrcpp::build_grid(f_tree, g_tree);    // Extend f with missing nodes relative to g
mrcpp::build_grid(f_tree, h_tree);    // Extend f with missing nodes relative to h
```

In contrast, doing the same with `copy_grid` would clear the `f_tree` grid in between, and you would *only* get a (identical) copy of the last `h_tree` grid, with no memory of the `g_tree` grid that was once there. One can also make the grids of two functions equal to their union

```
mrcpp::build_grid(f_tree, g_tree);    // Extend f with missing nodes relative to g
mrcpp::build_grid(g_tree, f_tree);    // Extend g with missing nodes relative to f
```

The union grid of several trees can be constructed in one go using a `FunctionTreeVector`

```
mrcpp::FunctionTreeVector<D> inp_vec;
inp_vec.push_back(std::make_tuple(1.0, tree_1));
inp_vec.push_back(std::make_tuple(1.0, tree_2));
inp_vec.push_back(std::make_tuple(1.0, tree_3));

mrcpp::FunctionTree<D> f_tree(MRA);
mrcpp::build_grid(f_tree, inp_vec);   // Extend f with missing nodes from all trees in␣
↪inp_vec
```

8.4.4 Projection

The `project` function takes an analytic D-dimensional scalar function (which can be defined as a lambda function or one of the explicitly implemented sub-classes of the `RepresentableFunction` base class in MRCPP) and projects it with the given precision onto the MRA defined by the `FunctionTree`. E.g. a unit charge Gaussian is projected in the following way (the MRA must be initialized as above)

```
// Defining an analytic function
double beta = 10.0;
double alpha = std::pow(beta/pi, 3.0/2.0);
auto func = [alpha, beta] (const mrcpp::Coord<3> &r) -> double {
    double R = std::sqrt(r[0]*r[0] + r[1]*r[1] + r[2]*r[2]);
    return alpha*std::exp(-beta*R*R);
};

double prec = 1.0e-5;
mrcpp::FunctionTree<3> tree(MRA);
mrcpp::project(prec, tree, func);
```

This projection will start at the default initial grid (only the root nodes of the given MRA), and adaptively build the full grid. Alternatively, the grid can be estimated *a priori* if the analytical function has some known features, such as for Gaussians:

```
double prec;                                    // Precision of the projection
int max_iter;                                   // Maximum levels of refinement

mrcpp::GaussFunc<D> func;                       // Analytic Gaussian function
mrcpp::FunctionTree<D> tree(MRA);               // Output tree

mrcpp::build_grid(tree, func);                  // Empty grid from analytic function
mrcpp::project(prec, tree, func, max_iter);     // Starts projecting from given grid
```

This will first produce an empty grid suited for representing the analytic function `func` (this is meant as a way to make sure that the projection starts on a grid where the function is actually visible, as for very narrow Gaussians, it's *not* meant to be a good approximation of the final grid) and then perform the projection on the given numerical grid. With a negative `prec` (or `max_iter = 0`) the projection will be performed strictly on the given initial grid, with no further refinements.

8.4.5 Addition

Arithmetic operations in the MW representation are performed using the `FunctionTreeVector`, and the general sum $f = \sum_i c_i f_i(x)$ is done in the following way

```
double a, b, c;                                    // Addition parameters
mrcpp::FunctionTree<D> a_tree(MRA);                // Input function
mrcpp::FunctionTree<D> b_tree(MRA);                // Input function
mrcpp::FunctionTree<D> c_tree(MRA);                // Input function

mrcpp::FunctionTreeVector<D> inp_vec;              // Vector to hold input functions
inp_vec.push_back(std::make_tuple(a, &a_tree));    // Append to vector
inp_vec.push_back(std::make_tuple(b, &b_tree));    // Append to vector
inp_vec.push_back(std::make_tuple(c, &c_tree));    // Append to vector

mrcpp::FunctionTree<D> f_tree(MRA);                // Output function
mrcpp::add(prec, f_tree, inp_vec);                 // Adaptive addition
```

The default initial grid is again only the root nodes, and a positive `prec` is required to build an adaptive tree structure for the result. The special case of adding two functions can be done directly without initializing a `FunctionTreeVector`

```
mrcpp::FunctionTree<D> f_tree(MRA);
mrcpp::add(prec, f_tree, a, a_tree, b, b_tree);
```

Addition of two functions is usually done on their (fixed) union grid

```
mrcpp::FunctionTree<D> f_tree(MRA);                // Construct empty root grid
mrcpp::build_grid(f_tree, a_tree);                 // Copy grid of g
mrcpp::build_grid(f_tree, b_tree);                 // Copy grid of h
mrcpp::add(-1.0, f_tree, a, a_tree, b, b_tree);    // Add functions on fixed grid
```

Note that in the case of addition there is no extra information to be gained by going beyond the finest refinement levels of the input functions, so the union grid summation is simply the best you can do, and adding a positive `prec` will not make a difference. There are situations where you want to use a *smaller* grid, though, e.g. when performing a unitary transformation among a set of `FunctionTrees`. In this case you usually don't want to construct *all* the output functions on the union grid of *all* the input functions, and this can be done by adding the functions adaptively starting from root nodes.

If you have a summation over several functions but want to perform the addition on the grid given by the *first* input function, you first copy the wanted grid and then perform the operation on that grid

```
mrcpp::FunctionTreeVector<D> inp_vec;
inp_vec.push_back(std::make_tuple(a, a_tree));
inp_vec.push_back(std::make_tuple(b, b_tree));
inp_vec.push_back(std::make_tuple(c, c_tree));

mrcpp::FunctionTree<D> f_tree(MRA);                // Construct empty root grid
mrcpp::copy_grid(f_tree, get_func(inp_vec, 0));    // Copy grid of first input function
mrcpp::add(-1.0, f_tree, inp_vec);                 // Add functions on fixed grid
```

Here you can of course also add a positive `prec` to the addition and the resulting function will be built adaptively starting from the given initial grid.

8.4.6 Multiplication

The multiplication follows the exact same syntax as the addition, where the product $f = \prod_i c_i f_i(x)$ is done in the following way

```
double a, b, c;                                    // Multiplication parameters
mrcpp::FunctionTree<D> a_tree(MRA);                // Input function
mrcpp::FunctionTree<D> b_tree(MRA);                // Input function
mrcpp::FunctionTree<D> c_tree(MRA);                // Input function

mrcpp::FunctionTreeVector<D> inp_vec;              // Vector to hold input functions
inp_vec.push_back(std::make_tuple(a, &a_tree));    // Append to vector
inp_vec.push_back(std::make_tuple(b, &b_tree));    // Append to vector
inp_vec.push_back(std::make_tuple(c, &c_tree));    // Append to vector

mrcpp::FunctionTree<D> f_tree(MRA);                // Output function
mrcpp::multipy(prec, f_tree, inp_vec);             // Adaptive multiplication
```

In the special case of multiplying two functions the coefficients are collected into one argument

```
mrcpp::FunctionTree<D> f_tree(MRA);
mrcpp::multiply(prec, f_tree, a*b, a_tree, b_tree);
```

For multiplications, there might be a loss of accuracy if the product is restricted to the union grid. The reason for this is that the product will contain signals of higher frequency than each of the input functions, which require a higher grid refinement for accurate representation. By specifying a positive `prec` you will allow the grid to adapt to the higher frequencies, but it is usually a good idea to restrict to one extra refinement level beyond the union grid (by setting `max_iter=1`) as the grids are not guaranteed to converge for such local operations (like arithmetics, derivatives and function mappings)

```
mrcpp::FunctionTree<D> f_tree(MRA);                    // Construct empty root grid
mrcpp::build_grid(f_tree, a_tree);                     // Copy grid of a
mrcpp::build_grid(f_tree, b_tree);                     // Copy grid of b
mrcpp::multiply(prec, f_tree, a*b, a_tree, b_tree, 1); // Allow 1 extra refinement
```

8.4.7 Re-using grids

Given a `FunctionTree` that is a valid function representation, we can clear its MW expansion coefficients as well as its grid refinement

```
mrcpp::FunctionTree<D> tree(MRA);                  // tree is an undefined function
mrcpp::project(prec, tree, f_func);                // tree represents analytic␣
↪function f
tree.clear();                                      // tree is an undefined function
mrcpp::project(prec, tree, f_func);                // tree represents analytic␣
↪function g
```

This action will leave the `FunctionTree` in the same state as after construction (undefined function, only root nodes), and its coefficients can now be re-computed.

In certain situations it might be desireable to separate the actions of computing MW coefficients and refining the grid. For this we can use the `refine_grid`, which will adaptively refine the grid one level (based on the wavelet norm and the given precision) and project the existing function representation onto the new finer grid

```
mrcpp::refine_grid(tree, prec);
```

E.i., this will *not* change the function that is represented in `tree`, but it *might* increase its grid size. The same effect can be made using another `FunctionTree` argument instead of the precision parameter

```
mrcpp::refine_grid(tree_out, tree_in);
```

which will *extend* the grid of `tree_out` in the same way as `build_grid` as shown above, but it will *keep* the function representation in `tree_out`.

This functionality can be combined with `clear_grid` to make a "manual" adaptive building algorithm. One example where this might be useful is in iterative algorithms where you want to fix the grid size for all calculations within one cycle and then relax the grid in the end in preparation for the next iteration. The following is equivalent to the adaptive projection above (`refine_grid` returns the number of new nodes that were created in the process)

```
int n_nodes = 1;
while (n_nodes > 0) {
    mrcpp::project(-1.0, tree, func);                  // Project f on fixed grid
    n_nodes = mrcpp::refine_grid(tree, prec);          // Refine grid based on prec
    if (n_nodes > 0) mrcpp::clear_grid(tree);          // Clear grid for next iteration
}
```

CHAPTER

NINE

MWOPERATORS

The MW operators discussed in this chapter is available to the application program by including:

```
#include "MRCPP/MWOperators"
```

9.1 ConvolutionOperator

Note: The convolution operators have separate precision parameters for their construction and application. The `build_prec` argument to the operator constructors will affect e.g. the number of terms in the separated representations of the Poisson/Helmholtz approximations, as well as the operator bandwidth. The `apply_prec` argument to the apply function relates only to the adaptive construction of the output function, based on a wavelet norm error estimate.

template<int **D**>

class **IdentityConvolution** : public mrcpp::ConvolutionOperator<*D*>

Convolution with an identity kernel.

The identity kernel (Dirac's delta function) is approximated by a narrow Gaussian function: $I(r - r') = \delta(r - r') \approx \alpha e^{-\beta(r-r')^2}$

Public Functions

IdentityConvolution(const *MultiResolutionAnalysis*<*D*> &mra, double prec)

This will project a kernel of a single gaussian with exponent sqrt(10/build_prec).

Parameters

- **mra** – **[in]** Which MRA the operator is defined
- **pr** – **[in]** Build precision, closeness to delta function

Returns
New *IdentityConvolution* object

template<int **D**>

class **DerivativeConvolution** : public mrcpp::ConvolutionOperator<*D*>

Convolution with a derivative kernel.

Derivative operator written as a convolution. The derivative kernel (derivative of Dirac's delta function) is approximated by the derivative of a narrow Gaussian function: $D^x(r - r') = \frac{d}{dx}\delta(r - r') \approx \frac{d}{dx}\alpha e^{-\beta(r-r')^2}$

NOTE: This is *not* the recommended derivative operator for practial calculations, it's a proof-of-concept operator. Use the *ABGVOperator* for "cuspy" functions and the *BSOperator* for smooth functions.

Public Functions

DerivativeConvolution(const *MultiResolutionAnalysis*<*D*> &mra, double prec)

This will project a kernel of a single differentiated gaussian with exponent sqrt(10/build_prec).

Parameters

- **mra** – **[in]** Which MRA the operator is defined
- **pr** – **[in]** Build precision, closeness to delta function

Returns
New *DerivativeConvolution* object

class **PoissonOperator** : public mrcpp::ConvolutionOperator<3>

Convolution with the Poisson Green's function kernel.

The Poisson kernel is approximated as a sum of Gaussian functions in order to allow for separated application of the operator in the Cartesian directions: $P(r-r') = \frac{1}{|r-r'|} \approx \sum_m^M \alpha_m e^{-\beta_m (r-r')^2}$

Public Functions

PoissonOperator(const *MultiResolutionAnalysis*<3> &mra, double prec)

This will construct a gaussian expansion to approximate 1/r, and project each term into a one-dimensional MW operator. Subsequent application of this operator will apply each of the terms to the input function in all Cartesian directions.

Parameters

- **mra** – **[in]** Which MRA the operator is defined
- **pr** – **[in]** Build precision, closeness to 1/r

Returns
New *PoissonOperator* object

class **HelmholtzOperator** : public mrcpp::ConvolutionOperator<3>

Convolution with the Helmholtz Green's function kernel.

The Helmholtz kernel is approximated as a sum of gaussian functions in order to allow for separated application of the operator in the Cartesian directions: $H(r-r') = \frac{e^{-\mu|r-r'|}}{|r-r'|} \approx \sum_m^M \alpha_m e^{-\beta_m (r-r')^2}$

Public Functions

HelmholtzOperator(const *MultiResolutionAnalysis*<3> &mra, double m, double prec)

This will construct a gaussian expansion to approximate exp(-mu*r)/r, and project each term into a one-dimensional MW operator. Subsequent application of this operator will apply each of the terms to the input function in all Cartesian directions.

Parameters

- **mra** – **[in]** Which MRA the operator is defined
- **m** – **[in]** Exponential parameter of the operator
- **pr** – **[in]** Build precision, closeness to exp(-mu*r)/r

Returns
New *HelmholtzOperator* object

template<int **D**>
void **mrcpp::apply**(double prec, *FunctionTree*<*D*> &out, ConvolutionOperator<*D*> &oper, *FunctionTree*<*D*> &inp, int maxIter, bool absPrec)

Application of MW integral convolution operator.

The output function will be computed using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **oper** – **[in]** Convolution operator to apply
- **inp** – **[in]** Input function
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree, default -1
- **absPrec** – **[in]** Build output tree based on absolute precision, default false

template<int **D**>
void **mrcpp::apply**(double prec, *FunctionTree*<*D*> &out, ConvolutionOperator<*D*> &oper, *FunctionTree*<*D*> &inp, FunctionTreeVector<*D*> &precTrees, int maxIter, bool absPrec)

Application of MW integral convolution operator.

The output function will be computed using the general algorithm:

- Compute MW coefs on current grid
- Refine grid where necessary based on *scaled* `prec`
- Repeat until convergence or `maxIter` is reached
- `prec < 0` or `maxIter = 0` means NO refinement
- `maxIter < 0` means no bound

The precision will be scaled locally by the maxNorms of the precTrees input vector.

Note: This algorithm will start at whatever grid is present in the `out` tree when the function is called (this grid should however be EMPTY, e.i. no coefs).

Parameters

- **prec** – **[in]** Build precision of output function
- **out** – **[out]** Output function to be built
- **oper** – **[in]** Convolution operator to apply
- **inp** – **[in]** Input function
- **precTrees** – **[in]** Precision trees
- **maxIter** – **[in]** Maximum number of refinement iterations in output tree, default -1
- **absPrec** – **[in]** Build output tree based on absolute precision, default false

9.2 DerivativeOperators

Note: The derivative operators have clearly defined requirements on the output grid structure, based on the grid of the input function. This means that there is no real grid adaptivity, and thus no precision parameter is needed for the application of such an operator.

---

template<int **D**>

class **ABGVOperator** : public mrcpp::DerivativeOperator<*D*>

Derivative operator as defined by Alpert, Beylkin, Ginez and Vozovoi, J Comp Phys 182, 149-190 (2002).

NOTE: This is the recommended derivative operator for "cuspy" or discontinuous functions. The *BSOperator* is recommended for smooth functions.

Public Functions

ABGVOperator(const *MultiResolutionAnalysis*<*D*> &mra, double a, double b)

Boundary parameters correspond to:

- `a=0.0 b=0.0`: Strictly local "center" difference
- `a=0.5 b=0.5`: Semi-local central difference
- `a=1.0 b=0.0`: Semi-local forward difference

- `a=0.0 b=1.0`: Semi-local backward difference

Parameters

- **mra** – **[in]** Which MRA the operator is defined
- **a** – **[in]** Left boundary condition
- **b** – **[in]** Right boundary condition

Returns
New *ABGVOperator* object

template<int **D**>

class **PHOperator** : public mrcpp::DerivativeOperator<*D*>

Derivative operator based on the smoothing derivative of Pavel Holoborodko .

NOTE: This is *not* the recommended derivative operator for practial calculations, it's a proof-of-concept operator. Use the *ABGVOperator* for "cuspy" functions and the *BSOperator* for smooth functions.

Public Functions

PHOperator(const *MultiResolutionAnalysis*<*D*> &mra, int order)

Parameters

- **mra** – **[in]** Which MRA the operator is defined
- **order** – **[in]** Derivative order, defined for 1 and 2

Returns
New *PHOperator* object

template<int **D**>

class **BSOperator** : public mrcpp::DerivativeOperator<*D*>

B-spline derivative operator as defined by Anderson etal, J Comp Phys X 4, 100033 (2019).

NOTE: This is the recommended derivative operator only for *smooth* functions. Use the *ABGVOperator* if the function has known cusps or discontinuities.

Public Functions

explicit **BSOperator**(const *MultiResolutionAnalysis*<*D*> &mra, int order)

Parameters

- **mra** – **[in]** Which MRA the operator is defined
- **order** – **[in]** Derivative order, defined for 1, 2 and 3

Returns
New *BSOperator* object

template<int **D**>

void mrcpp::**apply**(*FunctionTree*<*D*> &out, DerivativeOperator<*D*> &oper, *FunctionTree*<*D*> &inp, int dir)

Application of MW derivative operator.

The output function will be computed on a FIXED grid that is predetermined by the type of derivative operator. For a strictly local operator (ABGV_00), the grid is an exact copy of the input function. For operators that involve also neighboring nodes (ABGV_55, PH, BS) the base grid will be WIDENED by one node in the direction of application (on each side).

Note: The output function should contain only empty root nodes at entry.

Parameters

- **out** – **[out]** Output function to be built
- **oper** – **[in]** Derivative operator to apply
- **inp** – **[in]** Input function
- **dir** – **[in]** Direction of derivative

template<int **D**>
void mrcpp::**divergence**(*FunctionTree*<*D*> &out, DerivativeOperator<*D*> &oper, FunctionTreeVector<*D*> &inp)

Calculation of divergence of a function vector.

The derivative operator is applied in each Cartesian direction to the corresponding components of the input vector and added up to the final output. The grid of the output is fixed as the union of the component grids (including any derivative widening, see derivative apply).

Note:

- The length of the input vector must be the same as the template dimension D.
- The output function should contain only empty root nodes at entry.

Parameters

- **out** – **[out]** Output function
- **oper** – **[in]** Derivative operator to apply
- **inp** – **[in]** Input function vector

template<int **D**>
FunctionTreeVector<*D*> mrcpp::**gradient**(DerivativeOperator<*D*> &oper, *FunctionTree*<*D*> &inp)

Calculation of gradient vector of a function.

The derivative operator is applied in each Cartesian direction to the input function and appended to the output vector.

Note: The length of the output vector will be the template dimension D.

Parameters

- **oper** – **[in]** Derivative operator to apply
- **inp** – **[in]** Input function

Returns

FunctionTreeVector containing the gradient

9.3 Examples

9.3.1 PoissonOperator

The electrostatic potential g arising from a charge distribution f are related through the Poisson equation

$$-\nabla^2 g(r) = f(r)$$

This equation can be solved with respect to the potential by inverting the differential operator into the Green's function integral convolution operator

$$g(r) = \int \frac{1}{4\pi\|r - r'\|} f(r')dr'$$

This operator is available in the MW representation, and can be solved with arbitrary (finite) precision in linear complexity with respect to system size. Given an arbitrary charge dirtribution `f_tree` in the MW representation, the potential is computed in the following way:

```
double apply_prec;                                  // Precision for operator application
double build_prec;                                  // Precision for operator construction

mrcpp::PoissonOperator P(MRA, build_prec);          // MW representation of Poisson operator
mrcpp::FunctionTree<3> f_tree(MRA);                 // Input function
mrcpp::FunctionTree<3> g_tree(MRA);                 // Output function

mrcpp::apply(apply_prec, g_tree, P, f_tree);        // Apply operator adaptively
```

The Coulomb self-interaction energy can now be computed as the dot product:

```
double E = mrcpp::dot(g_tree, f_tree);
```

9.3.2 HelmholtzOperator

The Helmholtz operator is a generalization of the Poisson operator and is given as the integral convolution

$$g(r) = \int \frac{e^{-\mu\|r-r'\|}}{4\pi\|r - r'\|} f(r')dr'$$

The operator is the inverse of the shifted Laplacian

$$\left[-\nabla^2 + \mu^2\right] g(r) = f(r)$$

and appears e.g. when solving the SCF equations. The construction and application is similar to the Poisson operator, with an extra argument for the μ parameter

```
double apply_prec;                                  // Precision for operator application
double build_prec;                                  // Precision for operator construction
double mu;                                          // Must be a positive real number

mrcpp::HelmholtzOperator H(MRA, mu, build_prec);// MW representation of Helmholtz␣
↪operator
mrcpp::FunctionTree<3> f_tree(MRA);                 // Input function
mrcpp::FunctionTree<3> g_tree(MRA);                 // Output function

mrcpp::apply(apply_prec, g_tree, H, f_tree);        // Apply operator adaptively
```

9.3.3 ABGVOperator

The ABGV (Alpert, Beylkin, Gines, Vozovoi) derivative operator is initialized with two parameters a and b accounting for the boundary conditions between adjacent nodes, see Alpert et al.

```
double a = 0.0, b = 0.0;                            // Boundary conditions for operator
mrcpp::ABGVOperator<3> D(MRA, a, b);                // MW derivative operator
mrcpp::FunctionTree<3> f(MRA);                      // Input function
mrcpp::FunctionTree<3> f_x(MRA);                    // Output function
mrcpp::FunctionTree<3> f_y(MRA);                    // Output function
mrcpp::FunctionTree<3> f_z(MRA);                    // Output function

mrcpp::apply(f_x, D, f, 0);                         // Operator application in x direction
mrcpp::apply(f_y, D, f, 1);                         // Operator application in y direction
mrcpp::apply(f_z, D, f, 2);                         // Operator application in z direction
```

The tree structure of the output function will depend on the choice of parameters a and b: if both are zero, the output grid will be identical to the input grid; otherwise the grid will be widened by one node (on each side) in the direction of application.

9.3.4 PHOperator

The PH derivative operator is based on the noise reducing derivative of Pavel Holoborodko. This operator is also available as a direct second derivative.

```
mrcpp::PHOperator<3> D1(MRA, 1);                    // MW 1st derivative operator
mrcpp::PHOperator<3> D2(MRA, 2);                    // MW 2nd derivative operator
mrcpp::FunctionTree<3> f(MRA);                      // Input function
mrcpp::FunctionTree<3> f_x(MRA);                    // Output function
mrcpp::FunctionTree<3> f_xx(MRA);                   // Output function

mrcpp::apply(f_x, D1, f, 0);                        // Operator application in x direction
mrcpp::apply(f_xx, D2, f, 0);                       // Operator application in x direction
```

Special thanks to Prof. Robert J. Harrison (Stony Brook University) for sharing the operator coefficients.

9.3.5 BSOperator

The BS derivative operator is based on a pre-projection onto B-splines in order to remove the discontinuities in the MW basis, see Anderson et al. This operator is also available as a direct second and third derivative.

```
mrcpp::BSOperator<3> D1(MRA, 1);                // MW 1st derivative operator
mrcpp::BSOperator<3> D2(MRA, 2);                // MW 2nd derivative operator
mrcpp::BSOperator<3> D3(MRA, 3);                // MW 3nd derivative operator
mrcpp::FunctionTree<3> f(MRA);                  // Input function
mrcpp::FunctionTree<3> f_x(MRA);                // Output function
mrcpp::FunctionTree<3> f_yy(MRA);               // Output function
mrcpp::FunctionTree<3> f_zzz(MRA);              // Output function

mrcpp::apply(f_x, D1, f, 0);                    // Operator application in x direction
mrcpp::apply(f_yy, D2, f, 1);                   // Operator application in x direction
mrcpp::apply(f_zzz, D3, f, 2);                  // Operator application in x direction
```

CHAPTER

TEN

GAUSSIANS

MRCPP provides some simple features for analytic Gaussian functions. These are meant to be used as a starting point for MW computations, they are *not* meant for heavy analytical computation, like GTO basis sets. The Gaussian features are available by including:

```
#include "MRCPP/Gaussians"
```

template<int **D**>

class **GaussFunc** : public mrcpp::Gaussian<*D*>

Gaussian function in D dimensions with a simple monomial in front.

- Monodimensional Gaussian (GaussFunc<1>):
  $g(x) = \alpha(x - x_0)^a e^{-\beta(x-x_0)^2}$
- Multidimensional Gaussian (GaussFunc<D>):
  $G(x) = \prod_{d=1}^{D} g^d(x^d)$

Public Functions

inline **GaussFunc**(double beta, double alpha, const Coord<*D*> &pos = { }, const std::array<int, *D*> &pow = { })

Parameters

- **beta** – **[in]** Exponent, $e^{-\beta r^2}$
- **alpha** – **[in]** Coefficient, αe^{-r^2}
- **pos** – **[in]** Position $(x - pos[0]), (y - pos[1]), ...$
- **pow** – **[in]** Monomial power, $x^{pow[0]}, y^{pow[1]}, ...$

Returns
New *GaussFunc* object

double **calcCoulombEnergy**(const *GaussFunc*<*D*> &rhs) const

Compute Coulomb repulsion energy between two GaussFuncs.

Note: Both Gaussians must be normalized to unit charge $\alpha = (\beta/\pi)^{D/2}$ for this to be correct!

Parameters

- **this** – **[in]** Left hand *GaussFunc*
- **rhs** – **[in]** Right hand *GaussFunc*

Returns
Coulomb energy

virtual double **evalf**(const Coord<*D*> &r) const override

Parameters
r – **[in]** Cartesian coordinate

Returns
Function value in a point

virtual *GaussPoly*<*D*> **differentiate**(int dir) const override

Compute analytic derivative of Gaussian.

Parameters
dir – **[in]** Cartesian direction of derivative

Returns
New *GaussPoly*

GaussPoly<*D*> **mult**(const *GaussFunc*<*D*> &rhs)

Multiply two GaussFuncs.

Parameters

- **this** – **[in]** Left hand side of multiply
- **rhs** – **[in]** Right hand side of multiply

Returns
New *GaussPoly*

GaussFunc<*D*> **mult**(double c)

Multiply *GaussFunc* by scalar.

Parameters
c – **[in]** Scalar to multiply

Returns
New *GaussFunc*

GaussExp<*D*> **periodify**(const std::array<double, *D*> &period, double nStdDev = 4.0) const

Generates a *GaussExp* that is semi-periodic around a unit-cell.

nStdDev = 1, 2, 3 and 4 ensures atleast 68.27%, 95.45%, 99.73% and 99.99% of the integral is conserved with respect to the integration limits.

Parameters

- **period** – **[in]** The period of the unit cell
- **nStdDev** – **[in]** Number of standard diviations covered in each direction. Default 4.0

Returns
Semi-periodic version of a Gaussian around a unit-cell

inline void **normalize**()

Rescale function by its norm $||f||^{-1}$.

template<int **D**>

class **GaussPoly** : public mrcpp::Gaussian<*D*>

Gaussian function in D dimensions with a general polynomial in front.

- Monodimensional Gaussian (GaussPoly<1>):

  $g(x) = \alpha P(x - x_0) e^{-\beta(x-x_0)^2}$

- Multidimensional Gaussian (GaussFunc<D>):

  $G(x) = \prod_{d=1}^{D} g^d(x^d)$

Public Functions

GaussPoly(double alpha = 0.0, double coef = 1.0, const Coord<*D*> &pos = { }, const std::array<int, *D*> &power = { })

Parameters

- **beta** – **[in]** Exponent, $e^{-\beta r^2}$
- **alpha** – **[in]** Coefficient, αe^{-r^2}
- **pos** – **[in]** Position $(x - pos[0]), (y - pos[1]), ...$
- **pow** – **[in]** Max polynomial degree, $P_0(x), P_1(y), ...$

Returns

New *GaussPoly* object

virtual double **evalf**(const Coord<*D*> &r) const override

Parameters

r – **[in]** Cartesian coordinate

Returns

Function value in a point

virtual *GaussPoly* **differentiate**(int dir) const override

Compute analytic derivative of Gaussian.

Parameters

dir – **[in]** Cartesian direction of derivative

Returns

New *GaussPoly*

GaussPoly<*D*> **mult**(double c)

Multiply *GaussPoly* by scalar.

Parameters

c – **[in]** Scalar to multiply

Returns

New *GaussPoly*

void **setPoly**(int d, Polynomial &poly)

Set polynomial in given dimension.

Parameters

- **d** – **[in]** Cartesian direction
- **poly** – **[in]** Polynomial to set

GaussExp<*D*> **periodify**(const std::array<double, *D*> &period, double nStdDev = 4.0) const

Generates a *GaussExp* that is semi-periodic around a unit-cell.

nStdDev = 1, 2, 3 and 4 ensures atleast 68.27%, 95.45%, 99.73% and 99.99% of the integral is conserved with respect to the integration limits.

Parameters

- **period** – **[in]** The period of the unit cell
- **nStdDev** – **[in]** Number of standard diviations covered in each direction. Default 4.0

Returns

Semi-periodic version of a Gaussian around a unit-cell

inline void **normalize**()

Rescale function by its norm $||f||^{-1}$.

template<int **D**>

class **GaussExp** : public mrcpp::RepresentableFunction<*D*>

Gaussian expansion in D dimensions.

- Monodimensional Gaussian expansion:

  $g(x) = \sum_{m=1}^{M} g_m(x) = \sum_{m=1}^{M} \alpha_m e^{-\beta(x-x^0)^2}$
- Multidimensional Gaussian expansion:

  $G(x) = \sum_{m=1}^{M} G_m(x) = \sum_{m=1}^{M} \prod_{d=1}^{D} g_m^d(x^d)$

Public Functions

double **calcCoulombEnergy**() const

Note: Each Gaussian must be normalized to unit charge $c = (\alpha/\pi)^{D/2}$ for this to be correct!

Returns

Coulomb repulsion energy between all pairs in *GaussExp*, including self-interaction

virtual double **evalf**(const Coord<*D*> &r) const override

Parameters

r – **[in]** Cartesian coordinate

Returns

Function value in a point

void **append**(const Gaussian<*D*> &g)

Append Gaussian to expansion.

void **append**(const *GaussExp*<*D*> &g)

Append *GaussExp* to expansion.

10.1 Examples

A GaussFunc is a simple D-dimensional Gaussian function with a Cartesian monomial in front, e.g. in 3D:

$$f(r) = \alpha(x - x_0)^a(y - y_0)^b(z - z_0)^c e^{-\beta\|r-r_0\|^2}$$

```
double alpha, beta;
std::array<int, 3> pow = {a, b, c};
mrcpp::Coord<3> pos = {x_0, y_0, z_0};
mrcpp::GaussFunc<3> gauss(beta, alpha, pos, pow);

double E = gauss.calcCoulombEnergy(gauss);              // Analytical energy
```

This Gaussian function can be used to build an empty grid based on the position and exponent. The grid will then be refined close to the center of the Gaussian, with deeper refinement for higher exponents (steeper function):

```
mrcpp::FunctionTree<3> g_tree(MRA);
mrcpp::build_grid(g_tree, gauss);                       // Build empty grid
mrcpp::project(prec, g_tree, gauss);                    // Project Gaussian
```

GaussPoly is a generalization of the GaussFunc, where there is an arbitrary polynomial in front of the exponential

$$f(r) = \alpha P(r - r_0)e^{-\beta\|r-r_0\|^2}$$

For instance, the following function can be constructed

$$f(r) = \alpha(a_x + b_x x + c_x x^2)(a_y + b_y y + c_y y^2)(a_z + b_z z + c_z z^2)e^{-\beta\|r-r_0\|^2}$$

```
auto gauss_poly = GaussPoly<D>(beta, alpha, pos, pow);

// Create polynomial in x, y and z direction
auto pol_x = Polynomial(2); // 2 is the degree of the polynomial
pol_x.getCoefs() << a_x, b_x, c_x;
auto pol_y = Polynomial(2);
pol_y.getCoefs() << a_y, b_y, c_y;
auto pol_z = Polynomial(2);
pol_z.getCoefs() << a_z, b_z, c_z;

// Add polynomials to gauss_poly
guass_poly.setPoly(0, pol_x);
guass_poly.setPoly(1, pol_y);
guass_poly.setPoly(2, pol_z);
```

A GaussExp is a collection of Gaussians in the form

$$G(r) = \sum_i c_i g_i(r)$$

where g_i can be either `GaussFunc` or `GaussPoly`

$$g_i(r) = \alpha_i P_i(r - r_i) e^{-\beta_i \|r - r_i\|^2}$$

Individual Gaussian functions can be appended to the `GaussExp` and treated as a single function:

```
mrcpp::GaussExp<3> g_exp;                                   // Empty Gaussian expansion
for (int i = 0; i < N; i++) {
    double alpha_i, beta_i;                                 // Individual parameters
    std::array<int, 3> pow_i;                               // Individual parameters
    std::array<double, 3> pos_i;                            // Individual parameters
    mrcpp::GaussFunc<3> gauss_i(beta_i, alpha_i, pos_i, pow_i);
    g_exp.append(gauss_i);                                  // Append Gaussian to expansion
}
mrcpp::project(prec, tree, g_exp);                          // Project full expansion
```

CHAPTER

ELEVEN

PARALLEL

The core features of MRCPP are parallelized using a shared memory model *only* (OpenMP). This means that there is *no intrinsic* MPI parallelization (e.i. no data distribution across machines) *within* the library routines. However, the code comes with a small set of features that facilitate MPI work and data distribution in the host program, in the sense that *entire* `FunctionTree` objects can be located on different machines and communicated between them. Also, a `FunctionTree` can be *shared* between several MPI processes that are located on the *same* machine. This means that several processes have read access to the same `FunctionTree`, thus reducing the memory footprint, as well as the need for communication.

The MPI features are available by including:

```
#include "MRCPP/Parallel"
```

11.1 The host program

In order to utilize the MPI features of MRCPP, the MPI instance must be initialized (and finalized) by the host program, as usual:

```
MPI_Init(&argc, &argv);

int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size); // Get MPI world size
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get MPI world rank

MPI_Finalize();
```

For the shared memory features we must make sure that the ranks within a communicator is actually located on the same machine. When running on distributed architectures this can be achieved by creating separate communicators for each physical machine, e.g. to split *MPI_COMM_WORLD* into a new communicator group called *MPI_COMM_SHARED* that share the same physical memory space:

```
// Initialize a new communicator called MPI_COMM_SHARE
MPI_Comm MPI_COMM_SHARE;

// Split MPI_COMM_WORLD into sub groups and assign to MPI_COMM_SHARE
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &MPI_COMM_
↪SHARE);
```

Note that the main purpose of the shared memory feature of MRCPP is to avoid memory duplication and reduce the memory footprint, it will **not** automatically provide any work sharing parallelization for the construction of the shared `FunctionTree`.

11.2 Blocking communication

Warning: doxygenfunction: Unable to resolve function "mrcpp::send_tree" with arguments (FunctionTree<D>&, int, int, MPI_Comm, int) in doxygen xml output for project "MRCPP" from directory: _build/xml. Potential matches:

```
- template<int D> void send_tree(FunctionTree<D> &tree, int dst, int tag, mrcpp::mpi_
↪comm comm, int nChunks, bool coeff)
```

Warning: doxygenfunction: Unable to resolve function "mrcpp::recv_tree" with arguments (FunctionTree<D>&, int, int, MPI_Comm, int) in doxygen xml output for project "MRCPP" from directory: _build/xml. Potential matches:

```
- template<int D> void recv_tree(FunctionTree<D> &tree, int src, int tag, mrcpp::mpi_
↪comm comm, int nChunks, bool coeff)
```

11.2.1 Example

A blocking send/receive means that the function call does not return until the communication is completed. This is a simple and safe option, but can lead to significant overhead if the communicating MPI processes are not synchronized.

```
mrcpp::FunctionTree<3> tree(MRA);


// At this point tree is uninitialized on both rank 0 and 1


// Only rank 0 projects the function
if (rank == 0) mrcpp::project(prec, tree, func);


// At this point tree is projected on rank 0 but still uninitialized on rank 1


// Sending tree from rank 0 to rank 1
int tag = 111111; // Unique tag for each communication
int src=0, dst=1; // Source and destination ranks
if (rank == src) mrcpp::send_tree(tree, dst, tag, MPI_COMM_WORLD);
if (rank == dst) mrcpp::revc_tree(tree, src, tag, MPI_COMM_WORLD);


// At this point tree is projected on both rank 0 and 1


// Rank 0 clear the tree
if (rank == 0) mrcpp::clear(tree);


// At this point tree is uninitialized on rank 0 but still projected on rank 1
```

11.3 Shared memory

class **SharedMemory**

Shared memory block within a compute node.

This class defines a shared memory window in a shared MPI communicator. In order to allocate a *FunctionTree* in shared memory, simply pass a *SharedMemory* object to the *FunctionTree* constructor.

Public Functions

SharedMemory(mrcpp::mpi_comm comm, int sh_size)

SharedMemory constructor.

Parameters

- **comm** – **[in]** Communicator sharing resources
- **sh_size** – **[in]** Memory size, in MB

Warning: doxygenfunction: Unable to resolve function "mrcpp::share_tree" with arguments (FunctionTree<D>&, int, int, MPI_Comm) in doxygen xml output for project "MRCPP" from directory: _build/xml. Potential matches:

```
- template<int D> void share_tree(FunctionTree<D> &tree, int src, int tag, mrcpp::mpi_
↪comm comm)
```

11.3.1 Example

The sharing of a `FunctionTree` happens in three steps: first a `SharedMemory` object is initialized with the appropriate shared memory communicator; then this object is used in the `FunctionTree` constructor; finally, *after* the `FunctionTree` has been properly computed, a call must be made to the `share_tree` function. The reason for the last function call is that the internal memory pointers needs to be updated *locally* on each MPI process whenever the shared memory window has been updated.

```
// Get rank within the shared group
int rank;
MPI_Comm_rank(MPI_COMM_SHARE, &rank);

// Define master and worker ranks
int master = 0;
int worker = 1;

// The tree will be shared within the given communicator
int mem_size = 1000; //MB
mrcpp::SharedMemory shared_mem(MPI_COMM_SHARE, mem_size);
mrcpp::FunctionTree<3> tree(MRA, shared_mem);

// Master rank projects the tree
if (rank == master) mrcpp::project(prec, tree, func);

// When a shared function is updated, it must be re-shared
int tag = 333333; // Unique tag for each communication
```

(continues on next page)

(continued from previous page)

```
mrcpp::share_tree(tree, master, tag, MPI_COMM_SHARE);

// Other ranks within the shared group can update the tree
if (rank == worker) tree.rescale(2.0);

// When a shared function is updated, it must be re-shared
mrcpp::share_tree(tree, worker, tag, MPI_COMM_SHARE);
```

CHAPTER

TWELVE

PRINTER

MRCPP comes with a printer class to handle standard output:

```
#include "MRCPP/Printer"
```

The main purpose of this class is to provide (or suppress) any internal printing in MRCPP routines that might be useful for debugging. Also, it provides a sane printing environment for parallel computations that can also be used by the host program. By using the printing routines of this class, as opposed to the standard `std::cout`, only the master thread in a OpenMP region will provide any output while all other threads remain silent. Similarly, when running a host program in MPI parallel, the `mrcpp::Printer` provides three different options for handling printed output (see examples below):

- Only master rank prints to screen (stdout)
- All ranks prints to screen (stdout)
- All ranks prints to individual files

If you want only the master rank to print to an output file, this can be achieved by redirecting the output from the first option to a file (`./program >file.out`).

class **Printer**

Convenience class to handle printed output.

The *Printer* singleton class holds the current state of the print environment. All `mrcpp::print` functions, as well as the `println` and `printout` macros, take an integer print level as first argument. When the global *mrcpp::Printer* is initialized with a given print level, only print statements with a *lower* print level will be displayed. All internal printing in MRCPP is at print level 10 or higher, so there is some flexibility left (levels 0 through 9) for adjusting the print volume within the host program.

Public Static Functions

static void **init**(int level = 0, int rank = 0, int size = 1, const char *file = nullptr)

Initialize print environment.

Only print statements with lower printlevel than level will be displayed. If a file name is given, each process will print to a separate file called {file}-{rank}.out. If no file name is given, only processes which initialize the printer with rank=0 will print to screen. By default, all ranks initialize with rank=0, i.e. all ranks print to screen by default.

Parameters

- **level** – **[in]** Desired print level of output

- **rank** – **[in]** MPI rank of current process
- **size** – **[in]** Total number of MPI processes
- **file** – **[in]** File name for printed output, will get "-{rank}.out" extension

static inline void **setScientific**()

Use scientific floating point notation, e.g. 1.0e-2.

static inline void **setFixed**()

Use fixed floating point notation, e.g. 0.01.

static inline int **setWidth**(int i)

Set new line width for printed output.

Parameters
i – **[in]** New width (number of characters)

Returns
Old width (number of characters)

static inline int **setPrecision**(int i)

Set new precision for floating point output.

Parameters
i – **[in]** New precision (digits after comma)

Returns
Old precision (digits after comma)

static inline int **setPrintLevel**(int i)

Set new print level.

Parameters
i – **[in]** New print level

Returns
Old print level

static inline int **getWidth**()

Returns
Current line width (number of characters)

static inline int **getPrecision**()

Returns
Current precision for floating point output (digits after comma)

static inline int **getPrintLevel**()

Returns
Current print level

12.1 Functions

Some convenience functions for printing output is provided within the `mrcpp::print` namespace. These functions use the data of the `mrcpp::Printer` class to provide pretty output of a few standard data types.

void `mrcpp::print::environment`(int level)

Print information about MRCPP version and build configuration.

Parameters

`level` – **[in]** Activation level for print statement

void `mrcpp::print::separator`(int level, const char &c, int newlines = 0)

Print a full line of a single character.

Parameters

- **`level`** – **[in]** Activation level for print statement
- **`c`** – **[in]** Character to fill the line
- **`newlines`** – **[in]** Number of extra newlines

void `mrcpp::print::header`(int level, const std::string &txt, int newlines = 0, const char &c = '=')

Print a text header.

Parameters

- **`level`** – **[in]** Activation level for print statement
- **`txt`** – **[in]** Header text
- **`newlines`** – **[in]** Number of extra newlines
- **`c`** – **[in]** Character to fill the first line

void `mrcpp::print::footer`(int level, const *Timer* &timer, int newlines = 0, const char &c = '=')

Print a footer with elapsed wall time.

Parameters

- **`level`** – **[in]** Activation level for print statement
- **`t`** – **[in]** *Timer* to be evaluated
- **`newlines`** – **[in]** Number of extra newlines
- **`c`** – **[in]** Character to fill the last line

template<int **D**>
void `mrcpp::print::tree`(int level, const std::string &txt, const MWTree<*D*> &tree, const *Timer* &timer)

Print tree parameters (nodes, memory) and wall time.

Parameters

- **`level`** – **[in]** Activation level for print statement
- **`txt`** – **[in]** Text string
- **`tree`** – **[in]** Tree to be printed
- **`timer`** – **[in]** *Timer* to be evaluated

void mrcpp::print::**tree**(int level, const std::string &txt, int n, int m, double t)

Print tree parameters (nodes, memory) and wall time.

Parameters

- **level** – **[in]** Activation level for print statement
- **txt** – **[in]** Text string
- **n** – **[in]** Number of tree nodes
- **m** – **[in]** Memory usage (kB)
- **t** – **[in]** Wall time (sec)

void mrcpp::print::**time**(int level, const std::string &txt, const *Timer* &timer)

Print elapsed time from *Timer*.

Parameters

- **level** – **[in]** Activation level for print statement
- **txt** – **[in]** Text string
- **timer** – **[in]** *Timer* to be evaluated

void mrcpp::print::**memory**(int level, const std::string &txt)

Print the current memory usage of this process, obtained from system.

Parameters

- **level** – **[in]** Activation level for print statement
- **txt** – **[in]** Text string

12.2 Macros

The following macros should replace the regular calls to `std::cout`:

println(level, STR)

Print text at the given print level, with newline.

printout(level, STR)

Print text at the given print level, without newline.

The following macros will print a message along with information on where you are in the code (file name, line number and function name). Only macros that end with `_ABORT` will kill the program, all other will continue to run after the message is printed:

MSG_INFO(STR)

Print info message.

MSG_WARN(STR)

Print warning message.

MSG_ERROR(STR)

Print error message, no abort.

MSG_ABORT(STR)

Print error message and abort.

INVALID_ARG_ABORT

You have passed an invalid argument to a function.

NOT_IMPLEMENTED_ABORT

You have reached a point in the code that is not yet implemented.

NOT_REACHED_ABORT

You have reached a point that should not be reached, bug or inconsistency.

NEEDS_TESTING

You have reached an experimental part of the code, results cannot be trusted.

NEEDS_FIX(STR)

You have hit a known bug that is yet to be fixed, results cannot be trusted.

12.3 Examples

Using the print level to adjust the amount of output:

```
int level = 10;
mrcpp::Printer::init(level);            // Initialize printer with printlevel 10

println( 0, "This is printlevel  0");   // This will be displayed at printlevel 10
println(10, "This is printlevel 10");   // This will be displayed at printlevel 10
println(11, "This is printlevel 11");   // This will NOT be displayed at printlevel 10
```

Using headers and footers to get pretty output:

```
using namespace mrcpp;

Timer timer;                            // Start timer
project(prec, tree, func);              // Project function
double integral = tree.integrate();     // Integrate function
timer.stop();                           // Stop timer

print::header(0, "Projecting analytic function");
print::tree(0, "Projected function", tree, timer);
print::value(0, "Integrated function", integral, "(au)");
print::footer(0, timer);
```

This will produce the following output:

```
============================================================
               Projecting analytic function
------------------------------------------------------------
 Projected function         520 nds       16 MB    0.09 sec
 Integrated function               (au)  9.999999999992e-01
------------------------------------------------------------
                  Wall time: 9.32703e-02 sec
============================================================
```

As mentioned above, when running in MPI parallel there are three different ways of handling printed output (master to stdout, all to stdout or all to files). These can be chosen by adding appropriate arguments to `init`. The default setting will in a parallel environment have all MPI ranks printing to screen, but by adding MPI info to the printer, we can separate the output of the different ranks:

```
int level = 10;
int wrank, wsize;
MPI_Comm_rank(MPI_COMM_WORLD, &wrank);  // Get my rank
MPI_Comm_size(MPI_COMM_WORLD, &wsize);  // Get total number of ranks

// All ranks will print to screen
mrcpp::Printer::init(level);

// Only master rank will print to screen
mrcpp::Printer::init(level, wrank, wsize);

// All ranks will print to separate files called filename-<rank>.out
mrcpp::Printer::init(level, wrank, wsize, "filename");
```

CHAPTER

THIRTEEN

PLOTTER

MRCPP comes with its own plotter class which can be used by the host program to generate data files for visualization using e.g. gnuplot, paraview, blob and geomview. These features are available by including:

```
#include "MRCPP/Plotter"
```

template<int **D**>

class **Plotter**

Class for plotting multivariate functions.

This class will generate an equidistant grid in one (line), two (surf) or three (cube) dimensions, and subsequently evaluate the function on this grid.

The grid is generated from the vectors A, B and C, relative to the origin O:

- a linePlot will plot the line spanned by A, starting from O
- a surfPlot will plot the area spanned by A and B, starting from O
- a cubePlot will plot the volume spanned by A, B and C, starting from O

The vectors A, B and C do not necessarily have to be orthogonal.

The parameter `D` refers to the dimension of the *function*, not the dimension of the plot.

Public Functions

explicit **Plotter**(const Coord<*D*> &o = {})

Parameters
o – **[in]** Plot origin, default `(0, 0, ... , 0)`

Returns
New *Plotter* object

void **setSuffix**(int t, const std::string &s)

Set file extension for output file.

The file name you decide for the output will get a predefined suffix that differentiates between different types of plot.

Parameters

- **t** – **[in]** Plot type (*Plotter*<D>`::Line`, `::Surface`, `::Cube`, `::Grid`)
- **s** – **[in]** Extension string, default `.line`, `.surf`, `.cube`, `.grid`

void **setOrigin**(const Coord<*D*> &o)

Set the point of origin for the plot.

Parameters

o – **[in]** Plot origin, default `(0, 0, ... , 0)`

void **setRange**(const Coord<*D*> &a, const Coord<*D*> &b = {}, const Coord<*D*> &c = {})

Set boundary vectors A, B and C for the plot.

Parameters

- **a** – **[in]** A vector
- **b** – **[in]** B vector
- **c** – **[in]** C vector

void **gridPlot**(const MWTree<*D*> &tree, const std::string &fname)

Grid plot of a MWTree.

Writes a file named fname + file extension (".grid" as default) to be read by geomview to visualize the grid (of endNodes) where the multiresolution function is defined. In MPI, each process will write a separate file, and will print only nodes owned by itself (pluss the rootNodes).

Parameters

- **tree** – **[in]** MWTree to plot
- **fname** – **[in]** File name for output, without extension

void **linePlot**(const std::array<int, 1> &npts, const RepresentableFunction<*D*> &func, const std::string &fname)

Parametric plot of a function.

Plots the function func parametrically with npts[0] along the vector A starting from the origin O to a file named fname + file extension (".line" as default).

Parameters

- **npts** – **[in]** Number of points along A
- **func** – **[in]** Function to plot
- **fname** – **[in]** File name for output, without extension

void **surfPlot**(const std::array<int, 2> &npts, const RepresentableFunction<*D*> &func, const std::string &fname)

Surface plot of a function.

Plots the function func in 2D on the area spanned by the two vectors A (npts[0] points) and B (npts[1] points), starting from the origin O, to a file named fname + file extension (".surf" as default).

Parameters

- **npts** – **[in]** Number of points along A and B
- **func** – **[in]** Function to plot
- **fname** – **[in]** File name for output, without extension

void **cubePlot**(const std::array<int, 3> &npts, const RepresentableFunction<*D*> &func, const std::string &fname)

Cubic plot of a function.

Plots the function func in 3D in the volume spanned by the three vectors A (npts[0] points), B (npts[1] points) and C (npts[2] points), starting from the origin O, to a file named fname + file extension (".cube" as default).

Parameters

- **npts** – **[in]** Number of points along A, B and C
- **func** – **[in]** Function to plot
- **fname** – **[in]** File name for output, without extension

---

Note: When plotting a `FunctionTree`, only the *scaling* part of the leaf nodes will be evaluated, which means that the function values will not be fully accurate. This is done to allow a fast and `const` function evaluation that can be done in OMP parallel. If you want to include also the *final* wavelet corrections to your function values, you'll have to manually extend the MW grid by one level before plotting using `mrcpp::refine_grid(tree, 1)`.

13.1 Examples

A parametric line plot of a three-dimensional function along the z axis [-1, 1]:

```
mrcpp::FunctionTree<3> f_tree(MRA);                 // Function to be plotted

int nPts = 1000;                                    // Number of points
mrcpp::Coord<3> o{ 0.0, 0.0,-1.0};                  // Origin vector
mrcpp::Coord<3> a{ 0.0, 0.0, 2.0};                  // Boundary vector

mrcpp::Plotter<3> plot(o);                          // Plotter of 3D functions
plot.setRange(a);                                   // Set plot range
plot.linePlot({nPts}, f_tree, "f_tree");            // Write to file f_tree.line
```

A surface plot of a three-dimensional function in the x=[-2,2], y=[-1,1], z=0 plane:

```
int aPts = 2000;                                    // Number of points in a
int bPts = 1000;                                    // Number of points in b
mrcpp::Coord<3> o{-2.0,-1.0, 0.0};                  // Origin vector
mrcpp::Coord<3> a{ 4.0, 0.0, 0.0};                  // Boundary vector A
mrcpp::Coord<3> b{ 0.0, 2.0, 0.0};                  // Boundary vector B

mrcpp::Plotter<3> plot(o);                          // Plotter of 3D functions
plot.setRange(a, b);                                // Set plot range
plot.surfPlot({aPts, bPts}, f_tree, "f_tree");      // Write to file f_tree.surf
```

A cube plot of a three-dimensional function in the volume x=[-2,2], y=[-1,1], z=[0,2]:

```
int aPts = 200;                                     // Number of points in a
int bPts = 100;                                     // Number of points in b
```

(continues on next page)

(continued from previous page)

```
int cPts = 100;                                          // Number of points in c
mrcpp::Coord<3> o{-2.0,-1.0, 0.0};                       // Origin vector
mrcpp::Coord<3> a{ 4.0, 0.0, 0.0};                       // Boundary vector A
mrcpp::Coord<3> b{ 0.0, 2.0, 0.0};                       // Boundary vector B
mrcpp::Coord<3> b{ 0.0, 0.0, 2.0};                       // Boundary vector C

mrcpp::Plotter<3> plot(o);                               // Plotter of 3D functions
plot.setRange(a, b, c);                                  // Set plot range
plot.cubePlot({aPts, bPts, cPts}, f_tree, "f_tree"); // Write to file f_tree.cube
```

A grid plot of a three-dimensional FunctionTree:

```
mrcpp::Plotter<3> plot;                                  // Plotter of 3D functions
plot.gridPlot(f_tree, "f_tree");                         // Write to file f_tree.grid
```

CHAPTER

FOURTEEN

TIMER

MRCPP comes with a timer class which can be used by the host program:

```
#include "MRCPP/Timer"
```

class **Timer**

Records wall time between the execution of two lines of source code.

Public Functions

Timer(bool start_timer = true)

Parameters
start_timer – **[in]** option to start timer immediately

Returns
New *Timer* object

Timer(const *Timer* &timer)

Parameters
timer – **[in]** Object to copy

Returns
Copy of *Timer* object, including its current state

Timer &**operator=**(const *Timer* &timer)

Parameters
timer – **[in]** Object to copy

Returns
Copy of *Timer* object, including its current state

void **start**()

Start timer from zero.

void **resume**()

Resume timer from previous time.

void **stop**()

Stop timer.

double **elapsed**() const

Returns
Current elapsed time, in seconds

14.1 Examples

The timer records wall (human) time, not CPU user time. The clock will by default start immediately after construction, and will keep running until explicitly stopped. The elapsed time can be evaluated while clock is running:

```
mrcpp::Timer timer;                    // This will start the timer
mrcpp::project(prec, tree, func);      // Do some work
double t = timer.elapsed();            // Get time since clock started while still␣
↪running
```

The timer can also be started explicitly at a later stage *after* construction, as well as explicitly stopped after the work is done. Then the *elapsed()* function will return the time spent between *start()* and *stop()*:

```
mrcpp::Timer timer(false);             // This will not start the timer
timer.start();                         // This will start the timer
mrcpp::project(prec, tree, func);      // Do some work
timer.stop();                          // This will stop the timer
double t = timer.elapsed();            // Get time spent between start and stop
```

CHAPTER

FIFTEEN

PROGRAMMERS MANUAL

15.1 Clang-tidy

To ensure modern coding conventions are followed developers are encouraged to run clang-tidy on the code. Ensure clang-tidy is installed. Then to display available checkers run:

```
$ clang-tidy --list-checks -checks='*' | grep "modernize"
```

This will generate a list looking like this:

```
$ modernize-avoid-bind
$ modernize-deprecated-headers
$ modernize-loop-convert
$ modernize-make-shared
$ modernize-make-unique
$ modernize-pass-by-value
$ modernize-raw-string-literal
$ modernize-redundant-void-arg
$ modernize-replace-auto-ptr
$ modernize-replace-random-shuffle
$ modernize-return-braced-init-list
$ modernize-shrink-to-fit
$ modernize-unary-static-assert
$ modernize-use-auto
$ modernize-use-bool-literals
$ modernize-use-default-member-init
$ modernize-use-emplace
$ modernize-use-equals-default
$ modernize-use-equals-delete
$ modernize-use-noexcept
$ modernize-use-nullptr
$ modernize-use-override
$ modernize-use-transparent-functors
$ modernize-use-using
```

To run any of these modernization's on the code, go to your build directory. From there run the command:

```
$ run-clang-tidy -header-filter='.*' -checks='-*,modernize-your-modernization' -fix
```

INDEX

N

P