UltraCold

Generated by Doxygen 1.9.4

Chapter 1

UltraCold

1.1 About

UltraCold is a modular and extensible collection of C++ libraries for the study of ultra-cold atomic systems in the context of Gross-Pitaevskii theory.
The package contains several solver classes for different flavors of Gross-Pitaevskii and Bogolyubov equations, allowing for the description of ultra-cold systems of bosons at the mean-field level, studying their ground-state properties, the dynamics, and elementary excitations.

Right now, all the solver classes take advantage of OpenMP parallelization.

1.2 Prerequisites and platforms

UltraCold is built on top of Intel's Math Kernel Library, and relies upon `arpack-ng` (which is provided as a bundled package) for the solution of Bogolyubov equations. Hence, in order to use UltraCold, you first need to download and install a distribution of Intel's software.

Right now, the package has been only tested with Intel oneAPI, although it should also work with previous versions of Intel Parallel Studio. The Intel oneAPI package can be downloaded **for free** from `here`. In particular, UltraCold relies on the `Intel oneAPI Base Toolkit` and on the `Intel oneAPI HPC Toolkit`.

The package has been tested only on Linux machines, including the High Performance Computing cluster `Galileo100` from the italian supercomputing consortium `CINECA`.

1.3 Installation

To get UltraCold, first clone it into your machine
```
git clone https://github.com/smroccuzzo/UltraCold.git
```

Then, enter the directory UltraCold, and follow the usual steps required to build a project using `cmake` . By default, the build type is *Release*. So, all you have to do is
```
cd UltraCold
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=</your/install/path> ..
make
make install
```

1.4 Usage and examples

UltraCold comes packed with several solver classes for different flavors of Gross-Pitaevskii and Bogolyubov-like equations. The complete list of available solvers, as well as other useful classes (e.g. for data output) is available under the namespace UltraCold.
The Examples section provides several examples of usage of the basic functionalities of the package (as well as some physics). Such examples are meant to be used as templates for more complex applications, combining the functionality of the package to develop some interesting physics.

1.5 Contributing

UltraCold is developed using `Git` as a version control tool, and `GitHub` as the central host of the source code.

If you find some issue, and/or have suggestions for additions and/or improvements, please open a `GitHub issue`.

If you want to actively contribute, after opening an issue, follow this (pretty standard) workflow, based on the `fork and pull model`:

- If you already have a GitHub account, `sign in`. Otherwise, `create one` (it's free, and we are pretty confident that it will always be).
- Fork the UltraCold project.
- Make a local clone of your fork to your own computer.
- Create a new branch on which you will be making changes. This marks the point from which your copy of the project starts to differ from that of the main development branch.
- Start to make your changes, for example by modifying existing files and/or creating new ones. Once you are satisfied with your changes, you can commit each change, in such a way that Git can keep track of them. With each commit, write a short message describing what your particular set of changes does.
- When you're finished committing all of your changes to your local repository, you can push them all upstream to your GitHub repository.
- Finally, open a pull request on GitHub to the main development repository (i.e., by now, `this` one).

1.6 License

UltraCold is distributed as free software under the GPL3. See also the file LICENSE.md

Chapter 2

Examples

The UltraCold package contains several examples to illustrate how to use it to simulate ultra-cold atomic systems in the context of Gross-Pitaevskii theory.

The `examples` folder contains different folders called `example-<n>`, each containing

- a source file called `example-<n>.cpp`, containing an example on how to write an executable that uses the `UltraCold` library,
- a `CMakeLists.txt` containing instructions on how to configure and build an executable based on `UltraCold`,
- an eventual parameter file, called `example-<n>.prm`,

To run the examples, follow the usual steps required to build a project using `cmake`, namely, open a terminal in the folder containing the example you are interested in and type

```
mkdir build
cd build
cmake -DULTRACOLD_DIR=/path/to/the/directory/where/you/installed/UltraCold ..
make
```

This will create an executable called `example-<n>`, which (if everything went fine) should be ready to be executed. Now you can just copy the eventual file `example-<n>.prm` from the parent folder and run the example

```
cp ../example-<n>.prm .
./example-<n>
```

The output of course will depend on the particular example, and is documented fully for each of them.

Note

Although a prior basic knowledge of C++ is highly recommended, to use these examples also a very basic one is more than sufficient. The documentation tries to be as pedantic as possible, so that extending these examples for user's need shouldn't be too difficult.

Here is the complete list of all examples and a brief description of what each of them does. Refer to the detailed description available in the documentation.

- example-1 Defined in file `example-1.cpp` Ground state and simple dynamics of a three-dimensional, harmonically trapped Bose gas.
- example-2 Defined in file `example-2.cpp` Elementary excitations via Bogolyubov equations in a two-dimensional, harmonically trapped Bose gas.
- example-3 Defined in file `example-3.cpp` Supersolid ground state of a trapped dipolar Bose-Einstein condensate.
- example-4 Defined in file `example-4.cpp` Excitation spectrum of a trapped dipolar Bose-Einstein condensate across the superfluid-supersolid phase transition.

2.1 example-1

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

2.1.1 A three-dimensional Bose gas in a harmonic trap.

2.1.1.1 Introduction

In the first example, we are going to use UltraCold to study the ground state and a simple dynamics of a three-dimensional, harmonically trapped Bose gas of Rubidium atoms, using the solver class `UltraCold::GPSolvers::GPSolver`, or, better, **extending** this class for customizing the runtime behavior of real-time simulations. All the solver classes delivered with UltraCold can, in fact, be partially extended according to possible different needs, in particular for customizing the output of dynamic simulations or adding time-dependent terms to the Hamiltonian of the system to simulate, e.g., a ramp in the scattering length.

We will, in particular, solve the Gross-Pitaevskii equation on a three-dimensional, homogeneous mesh, calculating first the ground state of the system in the presence of an isotropic harmonic trap, using the member function `run_gradient_descent`, and then running a real-time dynamic simulation of an experiment for the measurement of the frequency of the so-called *breathing mode*, using the member function `run_operator_splitting`. The frequency of this collective oscillation can be calculated analytically in the Thomas-Fermi approximation, and is equal to $\omega_{breathing} = \sqrt{5}\omega_{ho}$. The breathing oscillation can be excited, for example, via a sudden isotropic shrinking of the harmonic trap. For the sake of illustration, however, we will use another procedure, namely a linear ramp in the scattering length, with a fixed time duration. This procedure, being isotropic in space, will still excite only the breathing mode we are interested in. It will also give us the opportunity to show how to derive a class from the base class `GPSolver` in order to introduce a time-dependent term in the Hamiltonian.

So, let's start from the Gross-Pitaevskii equation in three space dimensions, in the presence of an external harmonic potential

$$i\hbar\frac{\partial\psi(x,y,z,t)}{\partial t} = \left[\frac{-\hbar^2\nabla^2}{2m} + \frac{1}{2}m(\omega_x^2x^2+\omega_y^2y^2+\omega_z^2z^2) + \frac{4\pi a\hbar^2}{m}|\psi(x,y,z,t)|^2\right]\psi(x,y,z,t)$$

Measuring frequencies in units of the average harmonic frequency $\omega_{ho} = (\omega_x\omega_y\omega_z)^{\frac{1}{3}}$, lengths in units of the harmonic oscillator length $a_{ho} = \sqrt{\frac{\hbar}{m\omega_{ho}}}$, and times in units of ω_{ho}^{-1}, the equation can be recast in a-dimensional form as

$$i\frac{\partial\psi(x,y,z,t)}{\partial t} = \left[\frac{-\nabla^2}{2} + \frac{1}{2}(\omega_x^2x^2+\omega_y^2y^2+\omega_z^2z^2) + 4\pi a|\psi(x,y,z,t)|^2\right]\psi(x,y,z,t)$$

We will now see how to solve this equation for our needs using tools provided by UltraCold.

2.1.1.2 Program description

We first create an input file containing our mesh and physical parameters, as well as other parameters determining the run-time behavior of the system. Such input file will be called `example-1.prm` and contains the following text

```
# Mesh parameters

xmax = 10.0 # Size of the mesh along the x-axis, in micrometers. The mesh will extend from -xmax to xmax
ymax = 10.0 # Size of the mesh along the y-axis, in micrometers. The mesh will extend from -ymax to ymax
zmax = 10.0 # Size of the mesh along the z-axis, in micrometers. The mesh will extend from -zmax to zmax

nx = 64 # Number of points along the x-axis
ny = 64 # Number of points along the y-axis
nz = 64 # Number of points along the z-axis

# Physical parameters

initial scattering length   = 100.9  # Initial scattering length in units of the Bohr radius
number of particles         = 40000  # Total number of atoms
atomic mass                 = 87     # Atomic mass, in atomic mass units
omegax                      = 100 # Harmonic frequency along the x-axis, in units of (2pi)Hz
omegay                      = 100 # Harmonic frequency along the y-axis, in units of (2pi)Hz
omegaz                      = 100 # Harmonic frequency along the z-axis, in units of (2pi)Hz

# Run parameters for gradient descent

number of gradient descent steps = 10000 # maximum number of gradient descent steps
residual                         = 1.E-8 # Threshold on the norm of the residual
alpha                            = 1.E-4 # gradient descent step
beta                             = 0.9   # step for the heavy-ball acceleration method

# Run parameters for real-time dynamics

number of real time steps = 50000 # Total number of time-steps for real time dynamics
time step                 = 0.001 # Time step for real-time dynamics, in milliseconds
final scattering length   = 90.0  # Final scattering length in units of the Bohr radius
ramp duration             = 20.0  # Duration of the ramp of the scattering length, in milliseconds
```

We will see shortly how to read this file inside our `main()` function, using the class `UltraCold::Tools::InputParser`.

Let's take a look at the source code contained in the file `/examples/example-1/example-1.cpp`.

The first line includes the header file `"UltraCold.hpp"`, which allows to import all the interfaces to classes and functions available in the library.

```
#include "UltraCold.hpp"
```

The second line imports the namespace `UltraCold`, which wraps all the namespaces available in the library.

```
using namespace UltraCold;
```

As said, we will solve the Gross-Pitaevskii equation using a class derived from the `UltraCold::GPSolvers::GPSolver` class, which uses, under the hood, functions from Intel's Math Kernel Library to perform some mathematical operations, in particular dynamic allocation of data arrays and Fast Fourier Transforms. In the derived solver class, it is possible, among other things, to override the member functions `run_operator_splitting(...)`, which solves the Gross-Pitaevskii equation using operator splitting, `solve_step_1_operator_splitting(...)`, which solves the first step in the operator splitting method and allows to add time-dependent terms in the Hamiltonian, and `write_operator_splitting_output(...)`, which writes the output during dynamic simulations. So, let's define a custom `myGPSolver` class, inheriting from `UltraCold::GPSolvers::GPSolver`, and overriding the member functions described above:

```
class myGPSolver : public GPSolvers::GPSolver
{
    public:
        using GPSolver::GPSolver;
        void run_operator_splitting(int number_of_time_steps,
                                    double time_step,
                                    double ramp_duration,
                                    double initial_scattering_length,
                                    double final_scattering_length,
```

```
                                    std::ostream& output_stream) override;
        void write_operator_splitting_output(size_t iteration_number,
                                             double current_scattering_length,
                                             std::ostream& output_stream) override;
    protected:
        void solve_step_1_operator_splitting(double current_scattering_length) override;
};
```

First, we override `run_operator_splitting(...)` in such a way that it takes, as arguments, the duration of the ramp in the scattering length, as well as the values of the initial and final scattering lengths. The function will also perform a linear ramp in the scattering length during the real-time evolution of the system. Notice that the operator splitting procedure is here explicitly implemented, with the laplacian calculated using the Fast Fourier Transform routines from the Intel's Math Kernel Library, wrapped in the class `MKLWrappers::DFtCalculator`:

```
void myGPSolver::run_operator_splitting(int number_of_time_steps,
                                        double time_step,
                                        double ramp_duration,
                                        double initial_scattering_length,
                                        double final_scattering_length,
                                        std::ostream &output_stream)
{
    // Initialize the member variable time_step
    this->time_step = time_step;
    // Since the G.P. equation is solved on a cartesian mesh with periodic boundary conditions, a
    // DFtCalculator is needed to calculate the laplacian of psi
    MKLWrappers::DFtCalculator dft_calculator_step_2(psi,psitilde);
    //-----------------------------------------------------//
    //    Here the operator-splitting iterations start     //
    //-----------------------------------------------------//
    double current_scattering_length=initial_scattering_length;
    double current_time=0;
    for (size_t iteration_number = 0; iteration_number < number_of_time_steps; ++iteration_number)
    {
        // Write outputs starting from the first time step
        write_operator_splitting_output(iteration_number,
                                        current_scattering_length,
                                        output_stream);
        // Update the current value of the scattering length
        current_time = iteration_number*time_step;
        if(current_time <= ramp_duration)
        {
            std::cout « current_scattering_length*20361.7« std::endl;
            current_scattering_length = initial_scattering_length
                                        + (final_scattering_length-initial_scattering_length) *
       current_time/ramp_duration;
        }
        // Solve step 1 of operator splitting
        solve_step_1_operator_splitting(current_scattering_length);
        // Solve step 2 of operator splitting
        solve_step_2_operator_splitting(dft_calculator_step_2);
    }
}
```

Then, we also override the member function that solves the first step of the operator-splitting method in such a way that it uses the current value of the scattering length. Since we like to go fast, we also add a simple pre-processor directive instructing the compiler to parallelize the loop using OpenMP:

```
void myGPSolver::solve_step_1_operator_splitting(double current_scattering_length)
{
#pragma omp parallel for
        for (size_t i = 0; i < psi.size(); ++i)
            psi(i) *= std::exp(-ci*time_step*(Vext(i)+ 4*PI*current_scattering_length*std::norm(psi(i))));
}
```

Finally, we override the member function that writes the output of the real-time simulation, in such a way that it will calculate the root mean-squared radius of the atomic cloud every hundred time steps, writing it to the output stream together with the current time. Once again, since we like to go fast, we add a `#pragma` to parallelize the triple loop using OpenMP. Notice that both the time and the root mean squared radius will be in harmonic units:

```
void myGPSolver::write_operator_splitting_output(size_t iteration_number,
                                                 double current_scattering_length,
                                                 std::ostream &output_stream)
{
    if(iteration_number % 100 == 0)
    {
        double r2m = 0.0;
        double norm = 0.0;
#pragma omp parallel for reduction(+: r2m,norm) collapse(3)
        for (size_t i = 0; i < psi.extent(0); ++i)
            for(size_t j = 0; j < psi.extent(1); ++j)
                for (size_t k = 0; k < psi.extent(2); ++k)
                {
                    r2m += (std::pow(x[i],2)+std::pow(y[j],2)+std::pow(z[k],2))*std::norm(psi(i,j,k));
```

```
                    norm += std::norm(psi(i,j,k));
                }
        r2m = std::sqrt(r2m/norm);
        output_stream « iteration_number*time_step « " " « current_scattering_length « " " « r2m «
       std::endl;
    }
}
```

Now, we can define our main function.

```
int main() {
```

The first thing that main does is to define an object ot type `UltraCold::Tools::InputParser`, which allows to read the parameters defined in the file `example-1.prm` as follows:

```
Tools::InputParser ip("example-1.prm");
ip.read_input_file();
double xmax = ip.retrieve_double("xmax");
double ymax = ip.retrieve_double("ymax");
double zmax = ip.retrieve_double("zmax");
const int nx = ip.retrieve_int("nx");
const int ny = ip.retrieve_int("ny");
const int nz = ip.retrieve_int("nz");
double initial_scattering_length = ip.retrieve_double("initial scattering length");
const int    number_of_particles = ip.retrieve_int("number of particles");
const double atomic_mass         = ip.retrieve_double("atomic mass");
double omegax                    = ip.retrieve_double("omegax");
double omegay                    = ip.retrieve_double("omegay");
double omegaz                    = ip.retrieve_double("omegaz");
const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
const double residual                         = ip.retrieve_double("residual");
const double alpha                            = ip.retrieve_double("alpha");
const double beta                             = ip.retrieve_double("beta");
const int    number_of_real_time_steps = ip.retrieve_int("number of real time steps");
double time_step                       = ip.retrieve_double("time step");
double final_scattering_length         = ip.retrieve_double("final scattering length");
double ramp_duration                   = ip.retrieve_double("ramp duration");
```

Since it is very useful for comparison with typical scales used in experiments, in the input we gave lengths in micrometers, the scattering length in units of the Bohr radius, the time-step in milliseconds, and the atomic mass in atomic mass units. It is time to convert these parameters into harmonic units, in order to map them to an a-dimensional Gross-Pitaevskii equation:

```
const double hbar        = 0.6347*1.E5;
const double bohr_radius = 5.292E-5;
omegax *= TWOPI;
omegay *= TWOPI;
omegaz *= TWOPI;
const double omega_ho = std::cbrt(omegax*omegay*omegaz);
time_step     = time_step*omega_ho/1000.0;
ramp_duration = ramp_duration*omega_ho/1000.0;
omegax = omegax/omega_ho;
omegay = omegay/omega_ho;
omegaz = omegaz/omega_ho;
const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
initial_scattering_length *= bohr_radius/a_ho;
final_scattering_length   *= bohr_radius/a_ho;
xmax = xmax/a_ho;
ymax = ymax/a_ho;
zmax = zmax/a_ho;
```

We can now define the three-dimensional mesh on which the equation will be solved

```
Vector<double> x(nx);
Vector<double> y(ny);
Vector<double> z(nz);
double dx = 2.*xmax/nx;
double dy = 2.*ymax/ny;
double dz = 2.*zmax/nz;
for (size_t i = 0; i < nx; ++i) x(i) = -xmax + i*dx;
for (size_t i = 0; i < ny; ++i) y(i) = -ymax + i*dy;
for (size_t i = 0; i < nz; ++i) z(i) = -zmax + i*dz;
double dv = dx*dy*dz;
```

Next, we define an initial wave function, normalized to the total number of particles, and the external potential

```
Vector<std::complex<double>> psi(nx,ny,nz);
Vector<double> Vext(nx,ny,nz);
for (size_t i = 0; i < nx; ++i)
    for (size_t j = 0; j < ny; ++j)
        for (size_t k = 0; k < nz; ++k)
        {
            psi(i,j,k)  = exp(- (pow(x(i),2) +
                                 pow(y(j),2) +
                                 pow(z(k),2)) );
```

```
            Vext(i,j,k) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                std::pow(omegay,2)*pow(y(j),2) +
                                std::pow(omegaz,2)*pow(z(k),2) );
        }
double norm = 0.0;
for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
norm *= dv;
for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles/norm);
```

Finally, we initialize the myGPSolver class, and run the run_gradient_descent(...) member function in order to calculate a ground state solution on the defined mesh and for this external potential and physical parameters:

```
myGPSolver gp_solver(x,y,z,psi,Vext,initial_scattering_length);
std::fstream gradient_descent_output_stream;
gradient_descent_output_stream.open("gradient_descent_output.csv",std::ios::out);
double chemical_potential;
std::tie(psi,chemical_potential) = gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                  residual,
                                                                  alpha,
                                                                  beta,
                                                                  gradient_descent_output_stream);
gradient_descent_output_stream.close();
```

We write our ground state solution to a .vtk file, that can be read for plotting using programs like Paraview or Visit, using the class UltraCold::RealSpaceOut::DataOut:

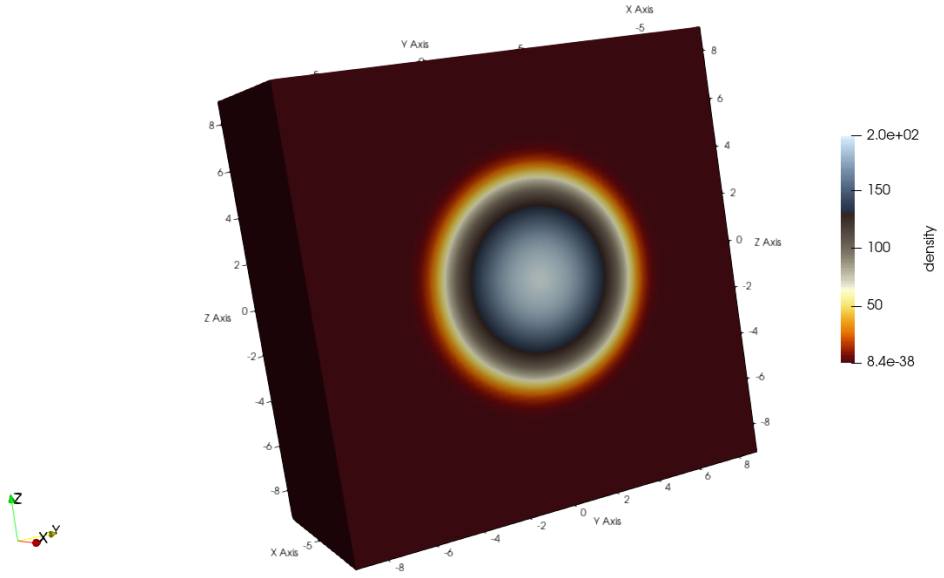
```
RealSpaceOutput::DataOut data_out;
data_out.set_output_name("ground_state_wave_function");
data_out.write_vtk(x,y,z,psi,"psi");
```

Finally, we re-initialize the solver, using as initial condition the ground state solution just calculated and run the dynamic simulation

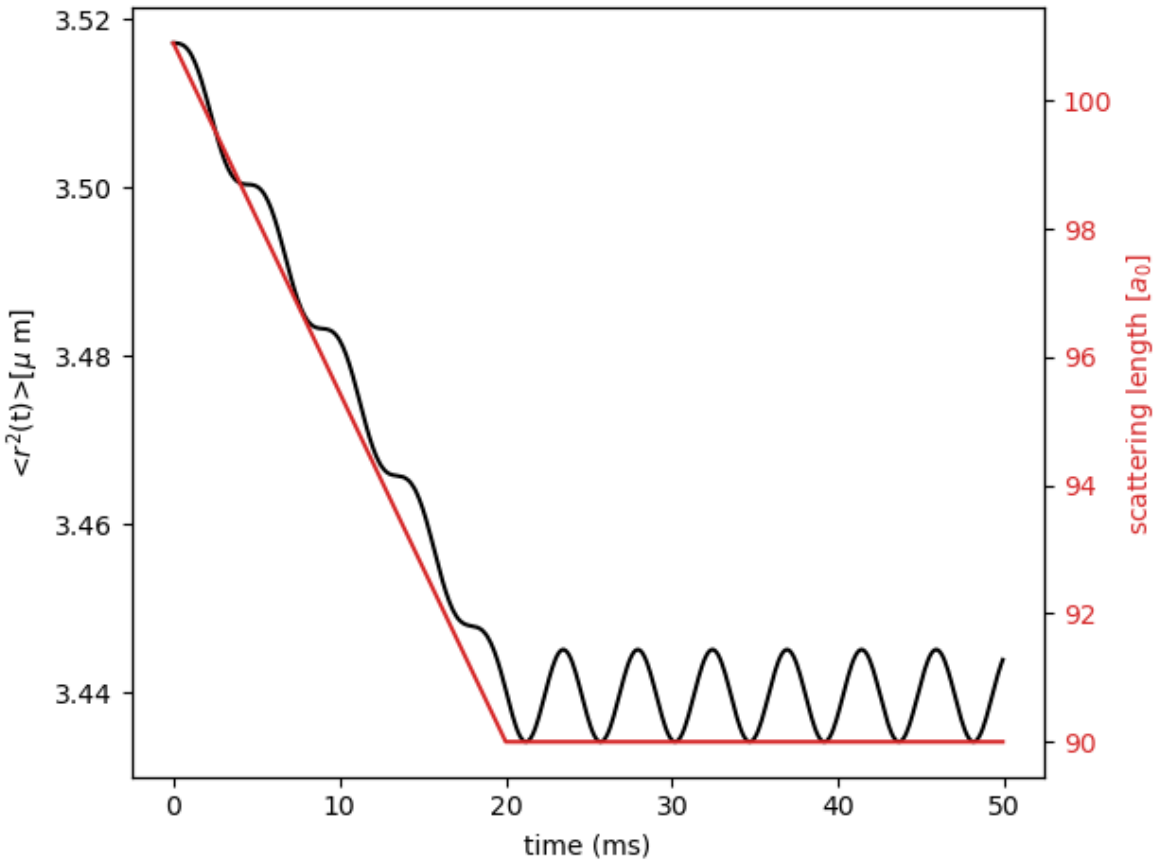
```
    gp_solver.reinit(Vext,psi);
    std::fstream output_file_stream;
    output_file_stream.open("real_time_output.csv",std::ios::out);
    gp_solver.run_operator_splitting(number_of_real_time_steps,
                                     time_step,
                                     ramp_duration,
                                     initial_scattering_length,
                                     final_scattering_length,
                                     output_file_stream);
    output_file_stream.close();
    return 0;
}
```

2.1.1.3 Results

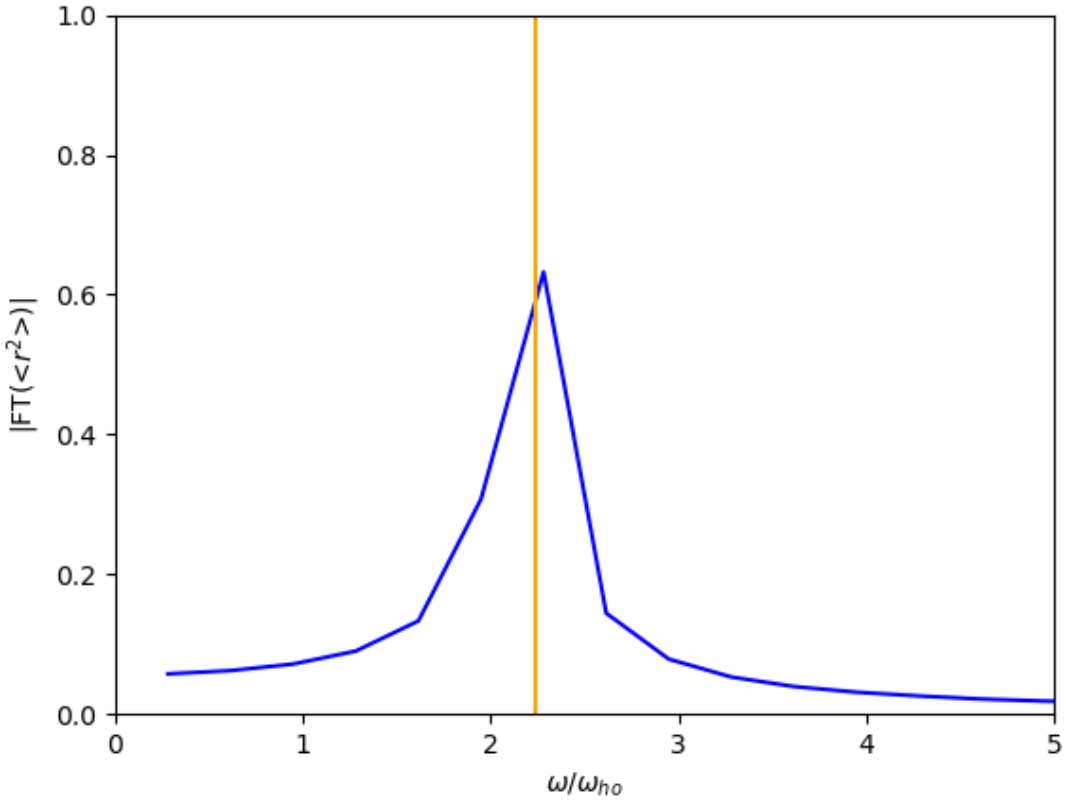
The ground state density profile can be visualized by opening the output file ground_state_wave_function.vtk, where we saved the mesh as well as the real and the imaginary part of the calculated ground-state wave function. Using, for example, Paraview , a typical output can look like the following



More interesting is the output of the real-time simulation dynamics. In fact, the value of squared mean radius of the cloud shows, as expected, a first decrease during the ramp, followed by a nice and simple harmonic oscillation:



The frequency of such harmonic oscillation can be extracted by a simple Fourier transform. The output looks like the following:



In this image, we also report the value of the frequency of the breathing mode calculated analytically, and corresponding, for this case, to $\sqrt{5}\omega_{ho}$. Such frequency is reported in the orange vertical line, and, as we can see, corresponds very well with the one extracted from the real time simulation.

2.1.1.4 Possible extensions

This program can be used as a template for studying the ground state and the dynamics of a simple BEC in different meshes and external potentials, or by exciting different collective oscillations, as e.g. the quadrupole mode.

2.1.1.5 The plain program

```
/*-----------------------------------------------------------------------------
 *
 *    This file is part of the UltraCold project.
 *
 *    UltraCold is free software: you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation, either version 3 of the License, or
 *    any later version.
 *    UltraCold is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *    You should have received a copy of the GNU General Public License
 *    along with UltraCold.  If not, see <https://www.gnu.org/licenses/>.
 *
 *-----------------------------------------------------------------------------*/
#include "UltraCold.hpp"
using namespace UltraCold;
class myGPSolver : public GPSolvers::GPSolver
{
```

```
    public:
        using GPSolver::GPSolver;
        void run_operator_splitting(int number_of_time_steps,
                                    double time_step,
                                    double ramp_duration,
                                    double initial_scattering_length,
                                    double final_scattering_length,
                                    std::ostream& output_stream) override;
        void write_operator_splitting_output(size_t iteration_number,
                                             double current_scattering_length,
                                             std::ostream& output_stream) override;
    protected:
        void solve_step_1_operator_splitting(double current_scattering_length) override;
};
void myGPSolver::run_operator_splitting(int number_of_time_steps,
                                        double time_step,
                                        double ramp_duration,
                                        double initial_scattering_length,
                                        double final_scattering_length,
                                        std::ostream &output_stream)
{
    // Initialize the member variable time_step
    this->time_step = time_step;
    // Since the G.P. equation is solved on a cartesian mesh with periodic boundary conditions, a
    // DFtCalculator is needed to calculate the laplacian of psi
    MKLWrappers::DFtCalculator dft_calculator_step_2(psi,psitilde);
    //-----------------------------------------------------//
    //    Here the operator-splitting iterations start     //
    //-----------------------------------------------------//
    double current_scattering_length=initial_scattering_length;
    double current_time=0;
    for (size_t iteration_number = 0; iteration_number < number_of_time_steps; ++iteration_number)
    {
        // Write outputs starting from the first time step
        write_operator_splitting_output(iteration_number,
                                        current_scattering_length,
                                        output_stream);
        // Update the current value of the scattering length
        current_time = iteration_number*time_step;
        if(current_time <= ramp_duration)
        {
            current_scattering_length = initial_scattering_length
                    + (final_scattering_length-initial_scattering_length) * current_time/ramp_duration;
        }
        // Solve step 1 of operator splitting
        solve_step_1_operator_splitting(current_scattering_length);
        // Solve step 2 of operator splitting
        solve_step_2_operator_splitting(dft_calculator_step_2);
    }
}
void myGPSolver::solve_step_1_operator_splitting(double current_scattering_length)
{
#pragma omp parallel for
        for (size_t i = 0; i < psi.size(); ++i)
            psi(i) *= std::exp(-ci*time_step*(Vext(i)+ 4*PI*current_scattering_length*std::norm(psi(i))));
}
void myGPSolver::write_operator_splitting_output(size_t iteration_number,
                                                 double current_scattering_length,
                                                 std::ostream &output_stream)
{
    if(iteration_number % 100 == 0)
    {
        double r2m = 0.0;
        double norm = 0.0;
#pragma omp parallel for reduction(+: r2m,norm) collapse(3)
        for (size_t i = 0; i < psi.extent(0); ++i)
            for(size_t j = 0; j < psi.extent(1); ++j)
                for (size_t k = 0; k < psi.extent(2); ++k)
                {
                    r2m += (std::pow(x[i],2)+std::pow(y[j],2)+std::pow(z[k],2))*std::norm(psi(i,j,k));
                    norm += std::norm(psi(i,j,k));
                }
        r2m = std::sqrt(r2m/norm);
        output_stream « iteration_number*time_step « " " « current_scattering_length « " " « r2m «
       std::endl;
    }
}
int main() {
    Tools::InputParser ip("../example-1.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    double zmax = ip.retrieve_double("zmax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    const int nz = ip.retrieve_int("nz");
    double initial_scattering_length = ip.retrieve_double("initial scattering length");
```

```
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    double omegaz                    = ip.retrieve_double("omegaz");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const int    number_of_real_time_steps = ip.retrieve_int("number of real time steps");
    double time_step                       = ip.retrieve_double("time step");
    double final_scattering_length         = ip.retrieve_double("final scattering length");
    double ramp_duration                   = ip.retrieve_double("ramp duration");
    const double hbar        = 0.6347*1.E5;
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    omegaz *= TWOPI;
    const double omega_ho = std::cbrt(omegax*omegay*omegaz);
    time_step     = time_step*omega_ho/1000.0;
    ramp_duration = ramp_duration*omega_ho/1000.0;
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    omegaz = omegaz/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    initial_scattering_length *= bohr_radius/a_ho;
    final_scattering_length   *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    zmax = zmax/a_ho;
    Vector<double> x(nx);
    Vector<double> y(ny);
    Vector<double> z(nz);
    double dx = 2.*xmax/nx;
    double dy = 2.*ymax/ny;
    double dz = 2.*zmax/nz;
    for (size_t i = 0; i < nx; ++i) x(i) = -xmax + i*dx;
    for (size_t i = 0; i < ny; ++i) y(i) = -ymax + i*dy;
    for (size_t i = 0; i < nz; ++i) z(i) = -zmax + i*dz;
    double dv = dx*dy*dz;
    Vector<std::complex<double>> psi(nx,ny,nz);
    Vector<double> Vext(nx,ny,nz);
    for (size_t i = 0; i < nx; ++i)
        for (size_t j = 0; j < ny; ++j)
            for (size_t k = 0; k < nz; ++k)
            {
                psi(i,j,k)  = exp(- (pow(x(i),2) +
                                     pow(y(j),2) +
                                     pow(z(k),2)) );
                Vext(i,j,k) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                    std::pow(omegay,2)*pow(y(j),2) +
                                    std::pow(omegaz,2)*pow(z(k),2) );
            }
    double norm = 0.0;
    for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
    norm *= dv;
    for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles/norm);
    myGPSolver gp_solver(x,y,z,psi,Vext,initial_scattering_length);
    std::fstream gradient_descent_output_stream;
    gradient_descent_output_stream.open("gradient_descent_output.csv",std::ios::out);
    double chemical_potential;
    std::tie(psi,chemical_potential) = gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                      residual,
                                                                      alpha,
                                                                      beta,
                                                                      gradient_descent_output_stream);
    gradient_descent_output_stream.close();
    RealSpaceOutput::DataOut data_out;
    data_out.set_output_name("ground_state_wave_function");
    data_out.write_vtk(x,y,z,psi,"psi");
    gp_solver.reinit(Vext,psi);
    std::fstream output_file_stream;
    output_file_stream.open("real_time_output.csv",std::ios::out);
    gp_solver.run_operator_splitting(number_of_real_time_steps,
                                     time_step,
                                     ramp_duration,
                                     initial_scattering_length,
                                     final_scattering_length,
                                     output_file_stream);
    output_file_stream.close();
    return 0;
}
```

2.2 example-2

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

2.2.1 Bogolyubov equations for a two-dimensional Bose gas in a harmonic trap.

2.2.1.1 Introduction

In the second example, we are going to use UltraCold to study the elementary excitations on top of the ground state of a two-dimensional, harmonically trapped Bose gas, by solving the so-called *Bogolyubov equations*. We will use the solver class `UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver`, which allows in fact to solve Bogolyubov equations for a simple Bose gas in the presence of an arbitrary external potential.

Starting from the a-dimensional Gross-Pitaevskii equation (GPe) in two space dimensions (assume we have already done some dimensional reduction integrating away the z-coordinate. This should lead to a renormalization of the scattering length, which we do not implement here because of laziness and because it does not affect, at least qualitatively, the physics) in the presence of an external harmonic potential

$$i\frac{\partial\psi(x,y,t)}{\partial t} = \left[\frac{-\nabla^2}{2} + \frac{1}{2}(\omega_x^2 x^2 + \omega_y^2 y^2) + 4\pi a|\psi(x,y,t)|^2\right]\psi(x,y,z,t)$$

one first searches for stationary solutions of the form

$$\psi(\mathbf{r},t) = \psi(\mathbf{r})e^{-i\frac{\mu}{\hbar}t}$$

obtaining the time-independent eigenvalue problem

$$\mu\psi(\mathbf{r}) = \left[\frac{-\nabla^2}{2} + \frac{1}{2}(\omega_x^2 x^2 + \omega_y^2 y^2) + 4\pi a|\psi(\mathbf{r})|^2\right]\psi(\mathbf{r})$$

The solution ψ_0 corresponding to the smallest eigenvalue μ is interpreted as the ground-state of the system, and the corresponding eigenvalue μ as the chemical potential.
In order to study the elementary excitations of the system on top of a certain ground-state solution, it is common to search for solutions of the time-dependent GPe of the form

$$\psi(\mathbf{r},t) = e^{-i\frac{\mu}{\hbar}t}\left[\psi_0(\mathbf{r}) + \sum_{n=0}^{\infty}\left(u_n(\mathbf{r})e^{-i\omega_n t} + v_n^*(\mathbf{r})e^{i\omega_n t}\right)\right]$$

Plugging this ansatz into the GPe and keeping only terms linear in the functions u and v, one obtains the following eigenvalue problem

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\frac{\nabla^2}{2} + V_{ext}(\mathbf{r}) + 4\pi a|\psi_0|^2 - \mu & 4\pi a\psi_0^2 \\ -4\pi a(\psi_0^*)^2 & -\left(-\frac{\nabla^2}{2} + V_{ext}(\mathbf{r}) + 4\pi a|\psi_0|^2 - \mu\right) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

which defines the so-called *Bogolyubov equations*. The solutions of these equations allows to study the energy of the elementary excitations of the system (*Bogolyubov modes*), as well as several interesting properties relative to the linear response of the system to external perturbations (see any book on Bose-Einstein condensation).

In the case in which the condensate ground-state wave function is real (e.g., in absence of vortices, solitons...) the problem can be recast in a more convenient form. In fact, taking the sum and the difference between the two equations, one easily finds

$$\hat{H}\hat{X}(u+v) = (\hbar\omega)^2(u+v)$$
$$\hat{X}\hat{H}(u-v) = (\hbar\omega)^2(u-v)$$

with

$$\hat{H} = -\frac{\nabla^2}{2} + V_{ext}(\mathbf{r}) + 4\pi a|\psi_0|^2 - \mu$$
$$\hat{X} = -\frac{\nabla^2}{2} + V_{ext}(\mathbf{r}) + 12\pi a|\psi_0|^2 - \mu$$

Now, both equations allow to find the (square) of the energy of the Bogolyubov modes, but solving a system of half the dimensionality of the original problem. This typically allows a great saving of computational time. The eigenvectors of the two problems correspond to $(u+v)$ and $(u-v)$ respectively, so that if one is interested in finding the Bogolyubov quasi-particle amplitudes u and v, one also needs to solve the second problem, and then set $u = 0.5\left((u+v)+(u-v)\right)$ and $v = 0.5\left((u+v)-(u-v)\right)$

2.2.1.2 Program description

For this example, we first need to calculate a ground state solution of the GPe for a certain set of physical and mesh parameter. Differently from example-1, we do not need anything special from this calculation, so we don't need to derive any class for our `UltraCold::GPSolvers::GPSolver` base class. Nonetheless, the input parameters describing the physics and the mesh can be practically the same as in example-1, except that we remove any reference to the z-axis. Our input file, called `example-2.prm`, will thus contain the following text

```
# Mesh parameters

xmax = 15.0 # Size of the mesh along the x-axis, in micrometers. The mesh will extend from -xmax to xmax
ymax = 15.0 # Size of the mesh along the y-axis, in micrometers. The mesh will extend from -ymax to ymax

nx = 128 # Number of points along the x-axis
ny = 128 # Number of points along the y-axis

# Physical parameters

scattering length   = 100.9  # Initial scattering length in units of the Bohr radius
number of particles = 40000  # Total number of atoms
atomic mass         = 87     # Atomic mass, in atomic mass units
omegax              = 100 # Harmonic frequency along the x-axis, in units of (2pi)Hz
omegay              = 100 # Harmonic frequency along the y-axis, in units of (2pi)Hz

# Run parameters for gradient descent

number of gradient descent steps = 50000 # maximum number of gradient descent steps
residual                         = 1.E-8 # Threshold on the norm of the residual
alpha                            = 1.E-4 # gradient descent step
beta                             = 0.9   # step for the heavy-ball acceleration method

# Run parameters for Bogolyubov equations

number of modes = 50
calculate eigenvectors = true
tolerance = 1.E-12
maximum number of arnoldi iterations = 1000
```

Let's now take a look at the source code contained in the file `/examples/example-2/example-2.cpp`.

The first lines again includes the header file `"UltraCold.hpp"` and import the namespace UltraCold

```
#include "UltraCold.hpp"
using namespace UltraCold;
```

As in example-1, we first calculate a ground-state solution of the GPe using the function `run_gradient_descent(...)` from the class UltraCold::GPSolvers::GPSolver. We don't go into much details here, since it is practically the same as in example-1

```
int main() {
    Tools::InputParser ip("../example-2.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    double       scattering_length   = ip.retrieve_double("scattering length");
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const int number_of_modes = ip.retrieve_int("number of modes");
    const int maximum_number_arnoldi_iterations = ip.retrieve_int("maximum number of arnoldi iterations");
    const double tolerance = ip.retrieve_double("tolerance");
    const bool calculate_eigenvectors = ip.retrieve_bool("calculate eigenvectors");
    const double hbar        = 0.6347*1.E5;
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    const double omega_ho = std::sqrt(omegax*omegay);
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    scattering_length   *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    Vector<double> x(nx);
    Vector<double> y(ny);
    double dx = 2.*xmax/nx;
    double dy = 2.*ymax/ny;
    for (size_t i = 0; i < nx; ++i) x(i) = -xmax + i*dx;
    for (size_t i = 0; i < ny; ++i) y(i) = -ymax + i*dy;
    double dv = dx*dy;
    Vector<std::complex<double>> psi(nx,ny);
    Vector<double> Vext(nx,ny);
    for (size_t i = 0; i < nx; ++i)
        for (size_t j = 0; j < ny; ++j)
            {
                psi(i,j)  = exp(- (pow(x(i),2) + pow(y(j),2)) );
                Vext(i,j) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                  std::pow(omegay,2)*pow(y(j),2));
            }
    double norm = 0.0;
    for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
    norm *= dv;
    for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles/norm);
    GPSolvers::GPSolver gp_solver(x,
                                  y,
                                  psi,
                                  Vext,
                                  scattering_length);
    double chemical_potential;
    std::tie(psi,chemical_potential) = gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                      residual,
                                                                      alpha,
                                                                      beta,
                                                                      std::cout);
    RealSpaceOutput::DataOut output_wave_function;
    output_wave_function.set_output_name("ground_state_wave_function");
    output_wave_function.write_vtk(x,y,psi,"ground_state_wave_function");
```

We can now plug the calculated ground-state solution ψ_0 to the class UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver, which will calculate for us the energies of the eigen-modes of the system as well as the Bogolyubov amplitudes u and v. Since we are considering a simple solution of the GPe, without any topological defect like solitons or vortices, the ground-state wave function, despite being defined as a complex Vector, will have only a non-zero real part. We can thus simplify the Bogolyubov equations solving only eigen-problems of halved dimensionality. The class `UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver` will do this automatically for

us, provided that we feed a real Vector, representing the ground-state wave function, to its constructor. We thus first copy the calculated ground-state wave-function into a real-valued Vector

```
Vector<double> psi0_real(nx,ny);
for (int i = 0; i < psi.size() ; ++i) psi0_real[i] = psi[i].real();
```

then initialize the data structures that will contain the solutions of the Bogolyubov equations

```
std::vector<std::complex<double» eigenvalues(number_of_modes);
std::vector< Vector<std::complex<double» > u(number_of_modes),v(number_of_modes);
```

and, finally, create our solver class and run the solver

```
BogolyubovSolvers::TrappedBogolyubovSolver bdg_solver(x,
                                                      y,
                                                      psi0_real,
                                                      Vext,
                                                      scattering_length,
                                                      chemical_potential,
                                                      number_of_modes,
                                                      tolerance,
                                                      maximum_number_arnoldi_iterations,
                                                      calculate_eigenvectors);
std::tie(eigenvalues,u,v) =  bdg_solver.run();
```

In the context of Bogolyubov theory, several interesting properties can be extracted from the knowledge of u and v. For example, one can see the density and phase fluctuations associated with each eigen-mode, by looking, respectively, at the quantities

$$\delta n(\mathbf{r}) = (u(\mathbf{r}) + v(\mathbf{r}))\psi_0(\mathbf{r})$$
$$\delta \phi(\mathbf{r}) = (u(\mathbf{r}) - v(\mathbf{r}))/\psi_0(\mathbf{r})$$

This is exactly what we calculated and output into some .vtk files with the last lines of the example

```
    RealSpaceOutput::DataOut                    output_fluctuations;
    std::vector<Vector< std::complex<double» > density_fluctuations(number_of_modes);
    std::vector<Vector< std::complex<double» > phase_fluctuations(number_of_modes);
    for (int i = 0; i < number_of_modes; ++i)
    {
        std::cout « eigenvalues[i].real() « " " « eigenvalues[i].imag() « std::endl;
        density_fluctuations[i].reinit(nx, ny);
        phase_fluctuations[i].reinit(nx, ny);
        for (int j = 0; j < nx * ny; ++j)
        {
            density_fluctuations[i](j) = (u[i](j) + v[i](j)) * psi0_real(j);
            phase_fluctuations[i](j)   = (u[i](j) - v[i](j)) / psi0_real(j);
        }
        output_fluctuations.set_output_name("density_fluctuations_mode_" + std::to_string(i));
        output_fluctuations.write_vtk(x, y, density_fluctuations[i], "density_fluctuations");
        output_fluctuations.set_output_name("phase_fluctuations_mode_" + std::to_string(i));
        output_fluctuations.write_vtk(x, y, phase_fluctuations[i], "phase_fluctuations");
    }
    return 0;
}
```

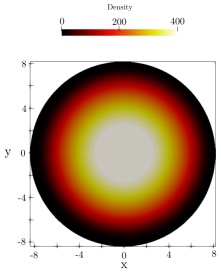
2.2.1.3 Results

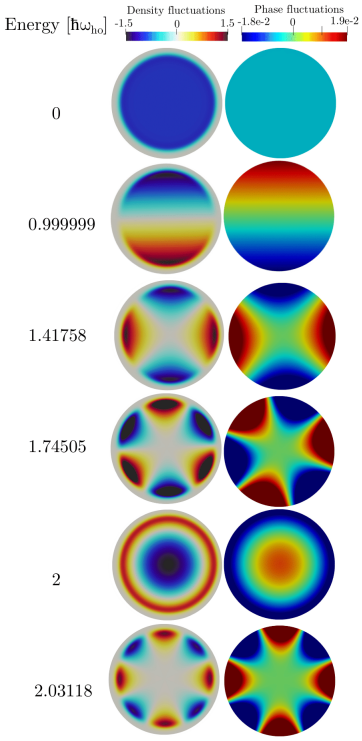
The output of the program related to the (first) calculated eigenvalues, corresponding to the energies of the Bogolyubov modes in units of $\hbar\omega_{ho}$, is the following:

```
1.6365e-06 0
0.999999 0
0.999999 0
1.41758 0
1.41758 0
1.74505 0
1.74505 0
2 0
2.03118 0
2.03118 0
2.29579 0
2.29579 0
```

Notice the presence of the two dipole modes, with a frequency approximately equal to one, representing the "sloshing" oscillations of the center of mass of the cloud along the x and y directions, and **always** present in the presence of harmonic trapping, as well as the presence of the characteristic *breathing* oscillation which, for this type of gas, has a frequency equal to $2\omega_{ho}$. Notice also that all the modes are twice degenerate.

We can also have a look at the ground-state density profile, as well as the density and phase fluctuations, plotted from the .vtk output using again Paraview:





We see that most of the modes are surface modes with different angular momenta, while the breathing mode (the fifth mode in order from top to bottom, with the frequency of $2\omega_{ho}$) has clearly a compressional character.

2.2.1.4 The plain program

```
/*-----------------------------------------------------------------------------
 *
 *    This file is part of the UltraCold project.
 *
 *    UltraCold is free software: you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation, either version 3 of the License, or
 *    any later version.
 *    UltraCold is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *    You should have received a copy of the GNU General Public License
 *    along with UltraCold.  If not, see <https://www.gnu.org/licenses/>.
 *
 *-----------------------------------------------------------------------------*/
#include "UltraCold.hpp"
using namespace UltraCold;
int main() {
    Tools::InputParser ip("../example-2.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    double       scattering_length   = ip.retrieve_double("scattering length");
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const int number_of_modes = ip.retrieve_int("number of modes");
    const int maximum_number_arnoldi_iterations = ip.retrieve_int("maximum number of arnoldi iterations");
    const double tolerance = ip.retrieve_double("tolerance");
    const bool calculate_eigenvectors = ip.retrieve_bool("calculate eigenvectors");
    const double hbar        = 0.6347*1.E5;
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    const double omega_ho = std::sqrt(omegax*omegay);
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    scattering_length   *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    Vector<double> x(nx);
    Vector<double> y(ny);
    double dx = 2.*xmax/nx;
    double dy = 2.*ymax/ny;
    for (size_t i = 0; i < nx; ++i) x(i) = -xmax + i*dx;
    for (size_t i = 0; i < ny; ++i) y(i) = -ymax + i*dy;
    double dv = dx*dy;
    Vector<std::complex<double>> psi(nx,ny);
    Vector<double> Vext(nx,ny);
    for (size_t i = 0; i < nx; ++i)
        for (size_t j = 0; j < ny; ++j)
            {
                psi(i,j)  = exp(- (pow(x(i),2) + pow(y(j),2)) );
                Vext(i,j) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                  std::pow(omegay,2)*pow(y(j),2));
            }
    double norm = 0.0;
    for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
    norm *= dv;
    for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles/norm);
    GPSolvers::GPSolver gp_solver(x,
                                  y,
                                  psi,
                                  Vext,
                                  scattering_length);
    double chemical_potential;
    std::tie(psi,chemical_potential) = gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                      residual,
                                                                      alpha,
                                                                      beta,
                                                                      std::cout);
    RealSpaceOutput::DataOut output_wave_function;
    output_wave_function.set_output_name("ground_state_wave_function");
    output_wave_function.write_vtk(x,y,psi,"ground_state_wave_function");
    Vector<double> psi0_real(nx,ny);
    for (int i = 0; i < psi.size() ; ++i) psi0_real[i] = psi[i].real();
```

```
    std::vector<std::complex<double» eigenvalues(number_of_modes);
    std::vector< Vector<std::complex<double» > u(number_of_modes),v(number_of_modes);
    BogolyubovSolvers::TrappedBogolyubovSolver bdg_solver(x,
                                                          y,
                                                          psi0_real,
                                                          Vext,
                                                          scattering_length,
                                                          chemical_potential,
                                                          number_of_modes,
                                                          tolerance,
                                                          maximum_number_arnoldi_iterations,
                                                          calculate_eigenvectors);
    std::tie(eigenvalues,u,v) =  bdg_solver.run();
    RealSpaceOutput::DataOut                    output_fluctuations;
    std::vector<Vector< std::complex<double» > density_fluctuations(number_of_modes);
    std::vector<Vector< std::complex<double» > phase_fluctuations(number_of_modes);
    for (int i = 0; i < number_of_modes; ++i)
    {
        std::cout « eigenvalues[i].real() « " " « eigenvalues[i].imag() « std::endl;
        density_fluctuations[i].reinit(nx, ny);
        phase_fluctuations[i].reinit(nx, ny);
        for (int j = 0; j < nx * ny; ++j)
        {
            density_fluctuations[i](j) = (u[i](j) + v[i](j)) * psi0_real(j);
            phase_fluctuations[i](j)   = (u[i](j) - v[i](j)) / psi0_real(j);
        }
        output_fluctuations.set_output_name("density_fluctuations_mode_" + std::to_string(i));
        output_fluctuations.write_vtk(x, y, density_fluctuations[i], "density_fluctuations");
        output_fluctuations.set_output_name("phase_fluctuations_mode_" + std::to_string(i));
        output_fluctuations.write_vtk(x, y, phase_fluctuations[i], "phase_fluctuations");
    }
    return 0;
}
```

2.3 example-3

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

2.3.1 A three-dimensional dipolar Bose gas in a harmonic trap.

2.3.1.1 Introduction

In this example, we are going to use UltraCold to study the ground state of a three-dimensional, harmonically trapped **dipolar** Bose gas of ^{164}Dy atoms, using the solver class `UltraCold::GPSolvers::DipolarGPSolver`.

Bose-Einstein condensates have been obtained in atomic species, like Erbium or Dysprosium , possessing a strong magnetic dipole moment in their ground state. This implies that, in order to describe the physics of such BECs, it is necessary to take into account the effect of magnetic interactions between the atoms. In the typical setup, atoms are aligned along a certain direction (say, the x-axis) by an external magnetic field, and their dipole-dipole interaction potential has the form

$$ V_{dd}({\bf r}-{\bf r}') = \frac{\mu_0\mu^2}{4\pi}\frac{1-3\cos^2(\theta)}{|{\bf r}-{\bf r}'|^3} $$

with μ_0 the magnetic permeability in vacuum, μ the magnetic dipole moment and θ the angle between the vector distance between dipoles and the polarization direction, i.e. in this case the x-axis. A simple mean-field description has been shown to fail in describing the observed properties of dipolar BECs. Currently, the most commonly used model for the description of dipolar BECs takes into account the first-order beyond mean-field correction to the ground-state energy of the system in the local density approximation. Such Lee-Huang-Yang (LHY) correction for a uniform system is given by

$$\frac{E_0}{V} = \frac{1}{2}gn^2\left[1 + \frac{128}{15\sqrt{\pi}}\sqrt{na^3}F(\epsilon_{dd})\right]$$

with

$$F(\epsilon_{dd}) = \frac{1}{2}\int_0^\pi d\theta sin\theta[1+\epsilon_{dd}(3cos^2\theta-1)]^{\frac{5}{2}}$$

Inserting this correction in the local density approximation in a mean-field model, we obtain the extended Gross-↩ Pitaevskii equation

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \mathcal{H}(\mathbf{r})\,\Psi(\mathbf{r},t)\,,$$

where the Hamiltonian H is

$$\mathcal{H}(\mathbf{r}) = -\frac{\hbar^2}{2m}\nabla^2 + V_{\rm ext}(\mathbf{r}) + g|\Psi(\mathbf{r},t)|^2 + \gamma(\varepsilon_{dd})|\Psi(\mathbf{r},t)|^3 + \int d\mathbf{r'}V_{dd}(\mathbf{r}-\mathbf{r'})|\Psi(\mathbf{r'},t)|^2\,,$$

with $g = 4\pi\hbar^2 a/m$ the coupling constant fixed by the s-wave scattering length a and

$$\gamma(\varepsilon_{dd}) = \frac{16}{3\sqrt{\pi}}ga^{\frac{3}{2}}\,\mathrm{Re}\left[\int_0^\pi d\theta\sin\theta[1+\varepsilon_{dd}(3\cos^2\theta-1)]^{\frac{5}{2}}\right].$$

In the absence of trapping, the system can be fully characterised by the single parameter $\varepsilon_{dd} = \mu_0\mu^2/(3g) = a_{dd}/a$, i.e., the ratio between the strength of the dipolar and the contact interaction, eventually written in terms of the dipolar length a_{dd} and the scattering length a.

Among the peculiar effects described by this model, we mention the possibility of describing the so-called quantum droplets , i.e. ultra-dilute, self-bound, liquid-like droplets in the Bose-Einstein condensed phase, and supersolids , i.e. phase-coherent systems spontaneously breaking translational invariance, developing spatial periodicity.

In this example, we will use the solver class `UltraCold::GPSolvers::DipolarGPSolver` to describe a simple supersolid state of a dipolar gas in a cigar-shaped harmonic trap.

2.3.1.2 Program description

We first create an input file containing our mesh and physical parameters, as well as other parameters determining the run-time behavior of the system. Such input file will be called `example-3.prm` and contains the following text

```
# Mesh parameters

xmax = 10.0 # Size of the mesh along the x-axis, in micrometers. The mesh will extend from -xmax to xmax
ymax = 10.0 # Size of the mesh along the y-axis, in micrometers. The mesh will extend from -ymax to ymax
zmax = 15.0 # Size of the mesh along the z-axis, in micrometers. The mesh will extend from -zmax to zmax

nx = 64 # Number of points along the x-axis
ny = 64 # Number of points along the y-axis
nz = 128 # Number of points along the z-axis

# Physical parameters

scattering length   = 95.0 # Scattering length in units of the Bohr radius
dipolar_length      = 132.0 # Dipolar length in units of the Bohr radius
number of particles = 40000  # Total number of atoms
atomic mass         = 164    # Atomic mass, in atomic mass units
omegax = 90 # Harmonic frequency along the x-axis, in units of (2pi)Hz
```

```
omegay = 60 # Harmonic frequency along the y-axis, in units of (2pi)Hz
omegaz = 30 # Harmonic frequency along the z-axis, in units of (2pi)Hz

# Run parameters for gradient descent

number of gradient descent steps = 20000 # maximum number of gradient descent steps
residual                         = 1.E-6 # Threshold on the norm of the residual
alpha                            = 1.E-3 # gradient descent step
beta                             = 0.9   # step for the heavy-ball acceleration method
```

Notice that the mesh is anisotropic, as well as the harmonic trap. In particular, we are using a trap elongated along the z-axis, and tighter along the x-axis. This is because the dipoles are aligned along the x-direction, and so, due to the partially attractive nature of the dipolar potential, they will try to "pile-up" in order to reach the lower energy attractive configuration with the dipoles sitting "head-to-tail". This can be partially prevented by using a tight harmonic trap along the polarization direction. Nonetheless, in the pure mean-field picture, using these parameters, the model would not admit any stable ground-state. Practically, solving the GPe without the LHY correction for the lowest energy state would result in a collapsed state, with the full wave-function concentrated in a single point of the mesh. Instead, the LHY correction will produce an interesting and stable ground-state configuration.

Let's look at the program. As in previous examples, we first read our input parameter file and set harmonic units

```
#include "UltraCold.hpp"
#include <random>
using namespace UltraCold;
int main()
{
    Tools::InputParser ip("example-3.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    double zmax = ip.retrieve_double("zmax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    const int nz = ip.retrieve_int("nz");
    double scattering_length         = ip.retrieve_double("scattering length");
    double dipolar_length            = ip.retrieve_double("dipolar_length");
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    double omegaz                    = ip.retrieve_double("omegaz");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const double hbar        = 0.6347*1.E5;
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    omegaz *= TWOPI;
    const double omega_ho = std::cbrt(omegax*omegay*omegaz);
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    omegaz = omegaz/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    scattering_length *= bohr_radius/a_ho;
    dipolar_length *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    zmax = zmax/a_ho;
```

Then, we define the mesh, the initial wave-function and the external potential. Notice that we add some random noise to the initial wave-function. This usually results in a speed-up of the convergence of the gradient-descent iterations

```
double dx = 2 * xmax / nx;
double dy = 2 * ymax / ny;
double dz = 2 * zmax / nz;
Vector<double> x(nx), y(ny), z(nz), kx(nx), ky(ny), kz(nz);
for (int i = 0; i < nx; ++i) x[i] = -xmax + i * dx;
for (int i = 0; i < ny; ++i) y[i] = -ymax + i * dy;
for (int i = 0; i < nz; ++i) z[i] = -zmax + i * dz;
create_mesh_in_Fourier_space(x, y, z, kx, ky, kz);
Vector<std::complex<double>> psi(nx, ny, nz);
Vector<double> Vext(nx, ny, nz);
std::default_random_engine generator;
std::uniform_real_distribution<double> distribution(0,1);
for (int i = 0; i < nx; ++i)
    for (int j = 0; j < ny; ++j)
```

```
        for (int k = 0; k < nz; ++k)
        {
            double random_number = distribution(generator);
            psi(i,j,k)  = (1.0+0.1*random_number)*
                    std::exp(-0.1*(pow(x(i),2) +
                                   pow(y(j),2) +
                                   pow(z(k),2)) );
            Vext(i,j,k) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                std::pow(omegay,2)*pow(y(j),2) +
                                std::pow(omegaz,2)*pow(z(k),2) );
        }
double norm = 0.0;
for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
norm *= (dx * dy * dz);
for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles / norm);
UltraCold::RealSpaceOutput::DataOut psi_out;
psi_out.set_output_name("initial_wave_function");
psi_out.write_slice2d_vtk(x,y,psi,"xy","initial_wave_function");
```

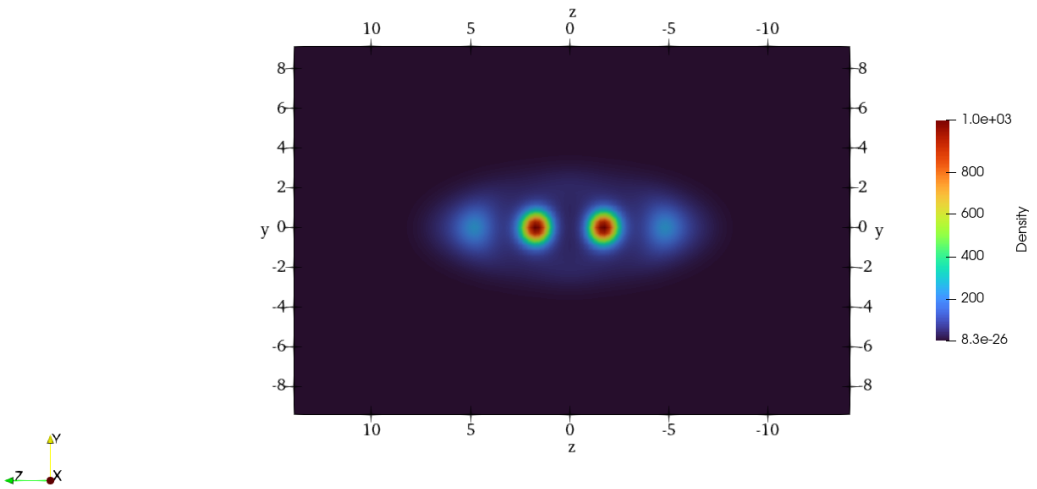
Next, we define our solver class, and run the solver for finding the ground state of our dipolar Bose gas

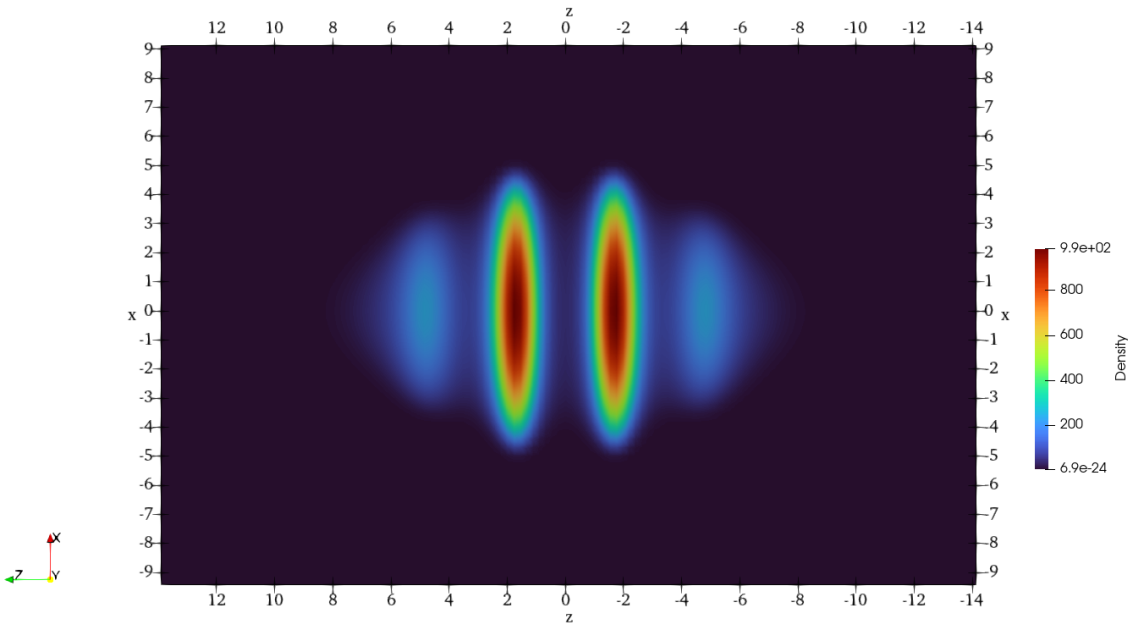
```
    GPSolvers::DipolarGPSolver dipolar_gp_solver(x,
                                                 y,
                                                 z,
                                                 psi,
                                                 Vext,
                                                 scattering_length,
                                                 dipolar_length);
    double chemical_potential;
    std::tie(psi, chemical_potential) =
       dipolar_gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                                residual,
                                                                                alpha,
                                                                                beta,
                                                                                std::cout);
    psi_out.set_output_name("ground_state_wave_function");
    psi_out.write_vtk(x,y,z,psi,"ground_state_wave_function");
    return 0;
}
```

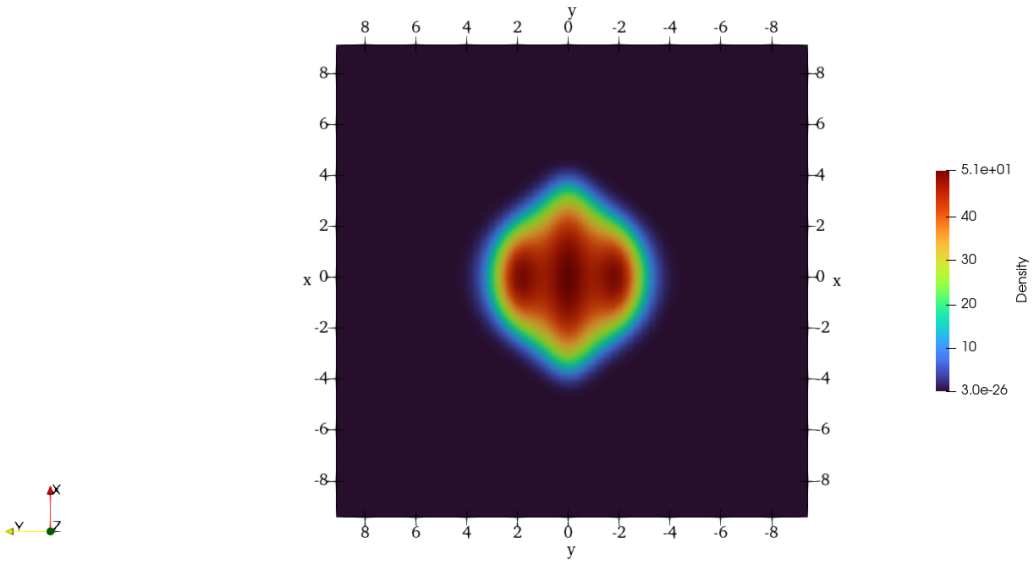
Let's get a look at the results.

2.3.1.3 Results

The ground state density profile can be visualized as usual by opening the output file `ground_state_wave_function.vtk`, where we saved the mesh as well as the real and the imaginary part of the calculated ground-state wave function. Using, as usual, `Paraview` , one can visualize slices of the ground-state density profiles along the three axis







Notice that the ground state density profile is characterized by the appearance of equally-spaced density peaks "immersed" in a more dilute superfluid background. Such states, characterized by a periodic density modulation of a phase-coherent system, are called **supersolids** and have been observed in experiments carried out, respectively, in Pisa , Innsbruck , and Stuttgart . For a recent review on the field, see e.g. here .

2.3.1.4 Possible extensions

This program can be extended just like example-1 for the study of the dynamics of the system, considering for example the transition from an ordinary superfluid to a supersolid by a ramp in the scattering length.

2.3.1.5 The plain program

```
/*-----------------------------------------------------------------------------
 *
 *    This file is part of the UltraCold project.
 *
 *    UltraCold is free software: you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation, either version 3 of the License, or
 *    any later version.
 *    UltraCold is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *    You should have received a copy of the GNU General Public License
 *    along with UltraCold.  If not, see <https://www.gnu.org/licenses/>.
 *
 *-----------------------------------------------------------------------------*/
#include "UltraCold.hpp"
#include <random>
using namespace UltraCold;
int main()
{
    Tools::InputParser ip("example-3.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    double zmax = ip.retrieve_double("zmax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    const int nz = ip.retrieve_int("nz");
    double scattering_length         = ip.retrieve_double("scattering length");
    double dipolar_length            = ip.retrieve_double("dipolar_length");
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    double omegaz                    = ip.retrieve_double("omegaz");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const double hbar        = 0.6347*1.E5;
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    omegaz *= TWOPI;
    const double omega_ho = std::cbrt(omegax*omegay*omegaz);
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    omegaz = omegaz/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    scattering_length *= bohr_radius/a_ho;
    dipolar_length *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    zmax = zmax/a_ho;
    double dx = 2 * xmax / nx;
    double dy = 2 * ymax / ny;
    double dz = 2 * zmax / nz;
    Vector<double> x(nx), y(ny), z(nz), kx(nx), ky(ny), kz(nz);
    for (int i = 0; i < nx; ++i) x[i] = -xmax + i * dx;
    for (int i = 0; i < ny; ++i) y[i] = -ymax + i * dy;
    for (int i = 0; i < nz; ++i) z[i] = -zmax + i * dz;
    create_mesh_in_Fourier_space(x, y, z, kx, ky, kz);
    Vector<std::complex<double>> psi(nx, ny, nz);
    Vector<double> Vext(nx, ny, nz);
    std::default_random_engine generator;
    std::uniform_real_distribution<double> distribution(0,1);
    for (int i = 0; i < nx; ++i)
        for (int j = 0; j < ny; ++j)
            for (int k = 0; k < nz; ++k)
            {
                double random_number = distribution(generator);
                psi(i,j,k)  = (1.0+0.1*random_number)*
                        std::exp(-0.1*(pow(x(i),2) +
                                       pow(y(j),2) +
                                       pow(z(k),2)) );
                Vext(i,j,k) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                    std::pow(omegay,2)*pow(y(j),2) +
                                    std::pow(omegaz,2)*pow(z(k),2) );
            }
    double norm = 0.0;
    for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
    norm *= (dx * dy * dz);
    for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles / norm);
    UltraCold::RealSpaceOutput::DataOut psi_out;
```

```
    psi_out.set_output_name("initial_wave_function");
    psi_out.write_slice2d_vtk(x,y,psi,"xy","initial_wave_function");
    GPSolvers::DipolarGPSolver dipolar_gp_solver(x,
                                                 y,
                                                 z,
                                                 psi,
                                                 Vext,
                                                 scattering_length,
                                                 dipolar_length);
    double chemical_potential;
    std::tie(psi, chemical_potential) =
       dipolar_gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                                residual,
                                                                                alpha,
                                                                                beta,
                                                                                std::cout);
    psi_out.set_output_name("ground_state_wave_function");
    psi_out.write_vtk(x,y,z,psi,"ground_state_wave_function");
    return 0;
}
```

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

2.3.2 A three-dimensional dipolar Bose gas in a harmonic trap.

2.3.2.1 Introduction

In this example, we are going to use UltraCold to study the ground state and of a three-dimensional, harmonically trapped **dipolar** Bose gas of ^{164}Dy atoms, using the solver class `UltraCold::GPSolvers::DipolarGPSolver`.

Bose-Einstein condensates have been obtained in atomic species, like Erbium or Dysprosium , possessing a strong magnetic dipole moment in their ground state. This implies that, in order to describe the physics of such BECs, it is necessary to take into account the effect of magnetic interactions between the atoms. In the typical setup, atoms are aligned along a certain direction (say, the x-axis) by an external magnetic field, and their dipole-dipole interaction potential has the form

$$V_{dd}(\mathbf{r}-\mathbf{r}') = \frac{\mu_0\mu^2}{4\pi}\frac{1-3\cos^2(\theta)}{|\mathbf{r}-\mathbf{r}'|^3}$$

with μ_0 the magnetic permeability in vacuum, μ the magnetic dipole moment and θ the angle between the vector distance between dipoles and the polarization direction, i.e. in this case the x-axis. A simple mean-field description has been shown to fail in describing the observed properties of dipolar BECs. Currently, the most commonly used model for the description of dipolar BECs takes into account the first-order beyond mean-field correction to the ground-state energy of the system in the local density approximation. Such Lee-Huang-Yang (LHY) correction for a uniform system is given by

$$\frac{E_0}{V} = \frac{1}{2}gn^2\left[1+\frac{128}{15\sqrt{\pi}}\sqrt{na^3}F(\epsilon_{dd})\right]$$

with

$$F(\epsilon_{dd}) = \frac{1}{2}\int_0^\pi d\theta sin\theta[1+\epsilon_{dd}(3cos^2\theta-1)]^{\frac{5}{2}}$$

Inserting this correction in the local density approximation in a mean-field model, we obtain the extended Gross-↩ Pitaevskii equation

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \mathcal{H}(\mathbf{r})\,\Psi(\mathbf{r},t)\,,$$

where the Hamiltonian H is

$$\mathcal{H}(\mathbf{r}) = -\frac{\hbar^2}{2m}\nabla^2 + V_{\text{ext}}(\mathbf{r}) + g|\Psi(\mathbf{r},t)|^2 + \gamma(\varepsilon_{dd})|\Psi(\mathbf{r},t)|^3 + \int d\mathbf{r}' V_{dd}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}',t)|^2\,,$$

with $g = 4\pi\hbar^2 a/m$ the coupling constant fixed by the s-wave scattering length a and

$$\gamma(\varepsilon_{dd}) = \frac{16}{3\sqrt{\pi}} g a^{\frac{3}{2}}\, \text{Re}\left[\int_0^\pi d\theta \sin\theta [1+\varepsilon_{dd}(3\cos^2\theta - 1)]^{\frac{5}{2}}\right].$$

In the absence of trapping, the system can be fully characterised by the single parameter $\varepsilon_{dd} = \mu_0\mu^2/(3g) = a_{dd}/a$, i.e., the ratio between the strength of the dipolar and the contact interaction, eventually written in terms of the dipolar length a_{dd} and the scattering length a.

Among the peculiar effects described by this model, we mention the possibility of describing the so-called quantum droplets , i.e. ultra-dilute, self-bound, liquid-like droplets in the Bose-Einstein condensed phase, and supersolids , i.e. phase-coherent systems spontaneously breaking translational invariance, developing spatial periodicity.

In this example, we will use the solver class `UltraCold::GPSolvers::DipolarGPSolver` to describe a simple supersolid state of a dipolar gas in a cigar-shaped harmonic trap.

2.3.2.2 Program description

We first create an input file containing our mesh and physical parameters, as well as other parameters determining the run-time behavior of the system. Such input file will be called `example-3.prm` and contains the following text

```
# Mesh parameters

xmax = 10.0 # Size of the mesh along the x-axis, in micrometers. The mesh will extend from -xmax to xmax
ymax = 10.0 # Size of the mesh along the y-axis, in micrometers. The mesh will extend from -ymax to ymax
zmax = 15.0 # Size of the mesh along the z-axis, in micrometers. The mesh will extend from -zmax to zmax

nx = 64 # Number of points along the x-axis
ny = 64 # Number of points along the y-axis
nz = 128 # Number of points along the z-axis

# Physical parameters

scattering length   = 95.0 # Scattering length in units of the Bohr radius
dipolar_length      = 132.0 # Dipolar length in units of the Bohr radius
number of particles = 40000  # Total number of atoms
atomic mass         = 164    # Atomic mass, in atomic mass units
omegax = 90 # Harmonic frequency along the x-axis, in units of (2pi)Hz
omegay = 60 # Harmonic frequency along the y-axis, in units of (2pi)Hz
omegaz = 30 # Harmonic frequency along the z-axis, in units of (2pi)Hz

# Run parameters for gradient descent

number of gradient descent steps = 20000 # maximum number of gradient descent steps
residual                         = 1.E-6 # Threshold on the norm of the residual
alpha                            = 1.E-3 # gradient descent step
beta                             = 0.9   # step for the heavy-ball acceleration method
```

Notice that the mesh is anisotropic, as well as the harmonic trap. In particular, we are using a trap elongated along the z-axis, and tighter along the x-axis. This is because the dipoles are aligned along the x-direction, and so, due to the partially attractive nature of the dipolar potential, they will try to "pile-up" in order to reach the lower energy attractive configuration with the dipoles sitting "head-to-tail". This can be partially prevented by using a tight harmonic trap along the polarization direction. Nonetheless, in the pure mean-field picture, using these parameters, the model would not admit any stable ground-state. Practically, solving the model without the LHY correction for the lowest energy state would result in a collapsed state, with the full wave-function concentrated in a single point of the mesh. Instead, the LHY correction will produce an interesting and stable ground-state configuration.

Let's look at the program. As in previous examples, we first read our input parameter file and set harmonic units

```
#include "UltraCold.hpp"
#include <random>
using namespace UltraCold;
int main()
{
    Tools::InputParser ip("example-3.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    double zmax = ip.retrieve_double("zmax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    const int nz = ip.retrieve_int("nz");
    double scattering_length         = ip.retrieve_double("scattering length");
    double dipolar_length            = ip.retrieve_double("dipolar_length");
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    double omegaz                    = ip.retrieve_double("omegaz");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const double hbar        = 0.6347*1.E5;
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    omegaz *= TWOPI;
    const double omega_ho = std::cbrt(omegax*omegay*omegaz);
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    omegaz = omegaz/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    scattering_length *= bohr_radius/a_ho;
    dipolar_length *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    zmax = zmax/a_ho;
```

Then, we define the mesh, the initial wave-function and the external potential. Notice that we add some random noise to the initial wave-function. This usually results in a speed-up of the convergence of the gradient-descent iterations

```
double dx = 2 * xmax / nx;
double dy = 2 * ymax / ny;
double dz = 2 * zmax / nz;
Vector<double> x(nx), y(ny), z(nz), kx(nx), ky(ny), kz(nz);
for (int i = 0; i < nx; ++i) x[i] = -xmax + i * dx;
for (int i = 0; i < ny; ++i) y[i] = -ymax + i * dy;
for (int i = 0; i < nz; ++i) z[i] = -zmax + i * dz;
create_mesh_in_Fourier_space(x, y, z, kx, ky, kz);
Vector<std::complex<double>> psi(nx, ny, nz);
Vector<double> Vext(nx, ny, nz);
std::default_random_engine generator;
std::uniform_real_distribution<double> distribution(0,1);
for (int i = 0; i < nx; ++i)
    for (int j = 0; j < ny; ++j)
        for (int k = 0; k < nz; ++k)
        {
            double random_number = distribution(generator);
            psi(i,j,k)  = (1.0+0.1*random_number)*
                    std::exp(-0.1*(pow(x(i),2) +
                                   pow(y(j),2) +
                                   pow(z(k),2)) );
            Vext(i,j,k) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                std::pow(omegay,2)*pow(y(j),2) +
                                std::pow(omegaz,2)*pow(z(k),2) );
        }
double norm = 0.0;
for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
norm *= (dx * dy * dz);
```

```
for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles / norm);
UltraCold::RealSpaceOutput::DataOut psi_out;
psi_out.set_output_name("initial_wave_function");
psi_out.write_slice2d_vtk(x,y,psi,"xy","initial_wave_function");
```

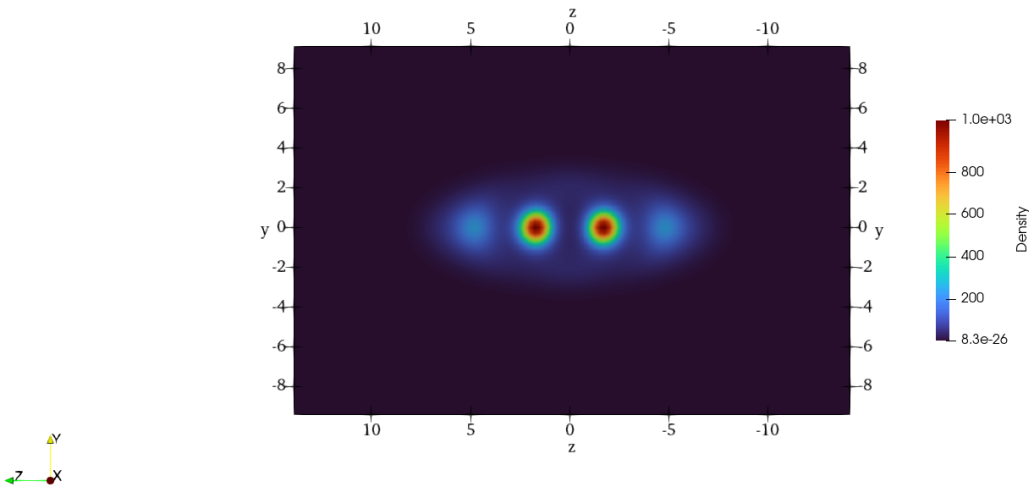
Next, we define our solver class, and run the solver for finding the ground state of our dipolar Bose gas

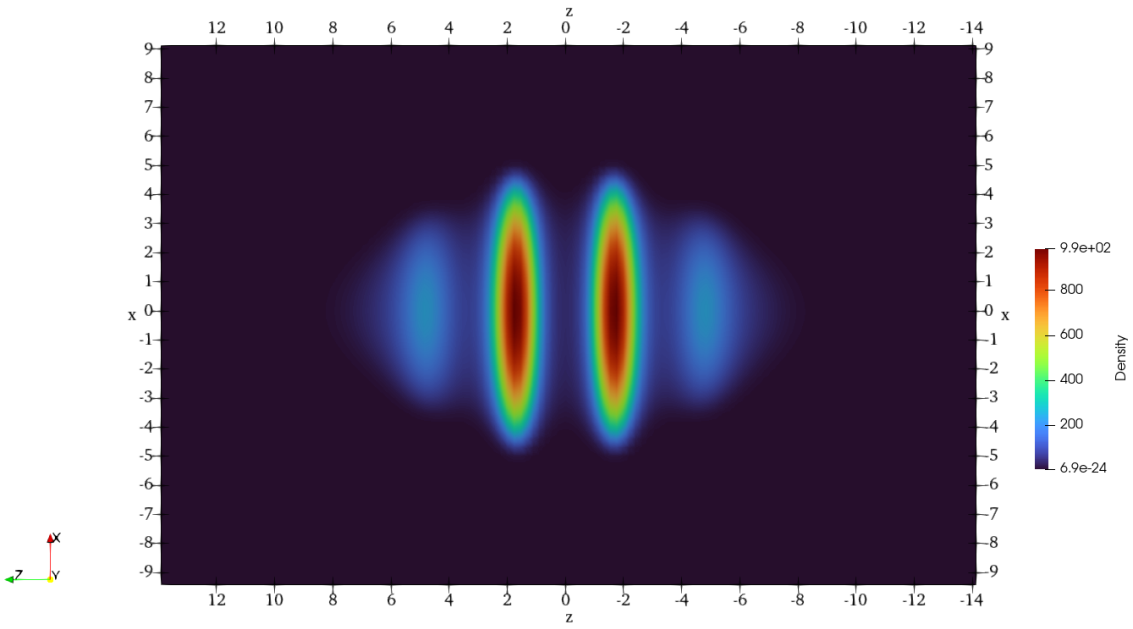
```
GPSolvers::DipolarGPSolver dipolar_gp_solver(x,
                                             y,
                                             z,
                                             psi,
                                             Vext,
                                             scattering_length,
                                             dipolar_length);
double chemical_potential;
std::tie(psi, chemical_potential) = dipolar_gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                           residual,
                                                                           alpha,
                                                                           beta,
                                                                           std::cout);
psi_out.set_output_name("ground_state_wave_function");
psi_out.write_vtk(x,y,z,psi,"ground_state_wave_function");
```

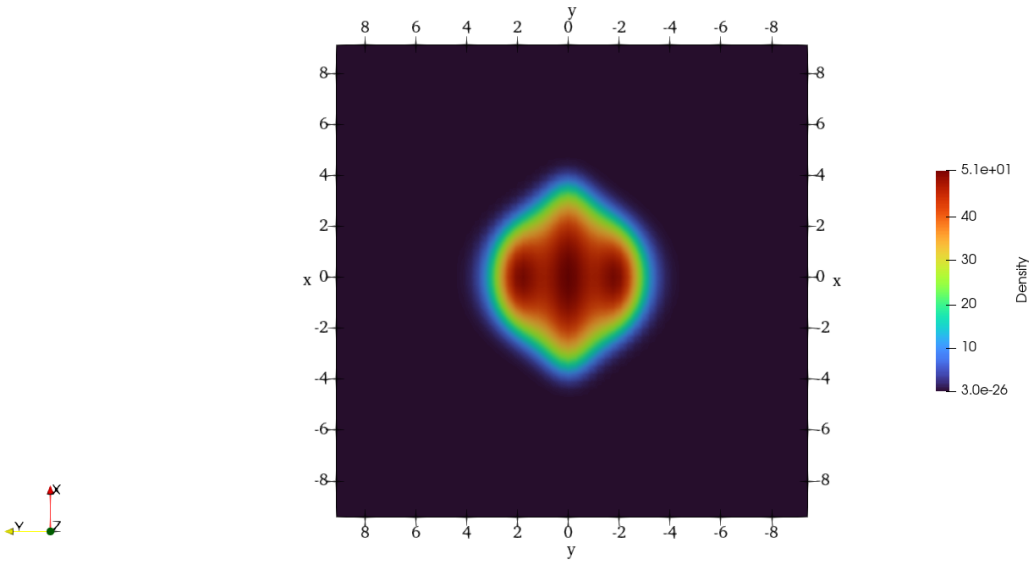
Let's get a look at the results.

2.3.2.3 Results

The ground state density profile can be visualized as usual by opening the output file `ground_state_wave_function.vtk`, where we saved the mesh as well as the real and the imaginary part of the calculated ground-state wave function. Using, as usual, `Paraview` , one can visualize slices of the ground-state density profiles along the three axis







Notice that the ground state density profile is characterized by the appearance of equally-spaced density peaks "immersed" in a more dilute superfluid background. Such states, characterized by a periodic density modulation of a phase-coherent system, are called **supersolids** and have been observed in experiments carried out, respectively, in Pisa , Innsbruck , and Stuttgart . For a recent review on the field, see e.g. here .

2.3.2.4 Possible extensions

This program can be extended just like example-1 for the study of the dynamics of the system, considering for example the transition from an ordinary superfluid to a supersolid by a ramp in the scattering length.

2.3.2.5 The plain program

```
/*-----------------------------------------------------------------------------
 *
 *    This file is part of the UltraCold project.
 *
 *    UltraCold is free software: you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation, either version 3 of the License, or
 *    any later version.
 *    UltraCold is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *    You should have received a copy of the GNU General Public License
 *    along with UltraCold.  If not, see <https://www.gnu.org/licenses/>.
 *
 *-----------------------------------------------------------------------------*/
#include "UltraCold.hpp"
#include <random>
using namespace UltraCold;
int main()
{
    Tools::InputParser ip("example-3.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    double zmax = ip.retrieve_double("zmax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    const int nz = ip.retrieve_int("nz");
    double scattering_length         = ip.retrieve_double("scattering length");
    double dipolar_length            = ip.retrieve_double("dipolar_length");
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    double omegaz                    = ip.retrieve_double("omegaz");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const double hbar        = 0.6347*1.E5;
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    omegaz *= TWOPI;
    const double omega_ho = std::cbrt(omegax*omegay*omegaz);
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    omegaz = omegaz/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    scattering_length *= bohr_radius/a_ho;
    dipolar_length *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    zmax = zmax/a_ho;
    double dx = 2 * xmax / nx;
    double dy = 2 * ymax / ny;
    double dz = 2 * zmax / nz;
    Vector<double> x(nx), y(ny), z(nz), kx(nx), ky(ny), kz(nz);
    for (int i = 0; i < nx; ++i) x[i] = -xmax + i * dx;
    for (int i = 0; i < ny; ++i) y[i] = -ymax + i * dy;
    for (int i = 0; i < nz; ++i) z[i] = -zmax + i * dz;
    create_mesh_in_Fourier_space(x, y, z, kx, ky, kz);
    Vector<std::complex<double>> psi(nx, ny, nz);
    Vector<double> Vext(nx, ny, nz);
    std::default_random_engine generator;
    std::uniform_real_distribution<double> distribution(0,1);
    for (int i = 0; i < nx; ++i)
        for (int j = 0; j < ny; ++j)
            for (int k = 0; k < nz; ++k)
            {
                double random_number = distribution(generator);
                psi(i,j,k)  = (1.0+0.1*random_number)*
                        std::exp(-0.1*(pow(x(i),2) +
                                       pow(y(j),2) +
                                       pow(z(k),2)) );
                Vext(i,j,k) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                    std::pow(omegay,2)*pow(y(j),2) +
                                    std::pow(omegaz,2)*pow(z(k),2) );
            }
    double norm = 0.0;
    for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
    norm *= (dx * dy * dz);
    for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles / norm);
    UltraCold::RealSpaceOutput::DataOut psi_out;
```

```
    psi_out.set_output_name("initial_wave_function");
    psi_out.write_slice2d_vtk(x,y,psi,"xy","initial_wave_function");
    GPSolvers::DipolarGPSolver dipolar_gp_solver(x,
                                                 y,
                                                 z,
                                                 psi,
                                                 Vext,
                                                 scattering_length,
                                                 dipolar_length);
    double chemical_potential;
    std::tie(psi, chemical_potential) =
       dipolar_gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                                residual,
                                                                                alpha,
                                                                                beta,
                                                                                std::cout);
    psi_out.set_output_name("ground_state_wave_function");
    psi_out.write_vtk(x,y,z,psi,"ground_state_wave_function");
    return 0;
}
```

2.4 example-4

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

2.4.1 Excitation spectrum of a trapped dipolar Bose-Einstein condensate across the superfluid-supersolid

phase transition.

2.4.1.1 Introduction

In this example, we are going to use UltraCold to study the elementary excitations of a three-dimensional, harmonically trapped **dipolar** Bose gas of ^{164}Dy atoms, using the solver class UltraCold::BogolyubovSolvers::TrappedDipol and across the superfluid-supersolid phase transition.

As explained in example-2, in order to calculate the spectrum of elementary excitations on top of a certain stationary solution of the GPe, it is necessary to search for solutions of the time-dependent GPe of the form

$$\psi(\mathbf{r},t) = e^{-i\frac{\mu}{\hbar}t}\left[\psi_0(\mathbf{r}) + \sum_{n=0}^{\infty}\left(u_n(\mathbf{r})e^{-i\omega_n t} + v_n^*(\mathbf{r})e^{i\omega_n t}\right)\right]$$

and solve the eigenvalue problem that comes out by keeping only terms linear in the quasi-particle amplitudes u and v. In the case of a **dipolar** Bose gas, taking also into account the effects of quantum fluctuations via the Lee-Huang-Yang (LHY) correction, this amounts to solving the following eigenvalue problem

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \hat{H} - \mu + \hat{X} & \hat{X}^\dagger \\ -\hat{X} & -(\hat{H} - \mu + \hat{X}^\dagger) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

with

$$\mathcal{H}(\mathbf{r}) = -\frac{\hbar^2}{2m}\nabla^2 + V_{\text{ext}}(\mathbf{r}) + g|\Psi(\mathbf{r},t)|^2 + \gamma(\varepsilon_{dd})|\Psi(\mathbf{r},t)|^3 + \int d\mathbf{r}' V_{dd}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}',t)|^2\,,$$

and

f(**r**) = $\psi_0(\mathbf{r}) \int d\mathbf{r}' V_{dd}(\mathbf{r}-\mathbf{r}') f(\mathbf{r}')\psi_0^*(\mathbf{r}') + \frac{3}{2}\gamma(\varepsilon_{dd})|\psi_0(\mathbf{r})|^3 f(\mathbf{r})$

and finally

$$\gamma(\varepsilon_{dd}) = \frac{16}{3\sqrt{\pi}} g a^{\frac{3}{2}}\, \text{Re}\left[\int_0^\pi d\theta \sin\theta [1+\varepsilon_{dd}(3\cos^2\theta - 1)]^{\frac{5}{2}}\right].$$

with $g = 4\pi\hbar^2 a/m$ the coupling constant fixed by the s-wave scattering length a, $V_{dd}(\mathbf{r}_i - \mathbf{r}_j) = \frac{\mu_0\mu^2}{4\pi}\frac{1-3\cos^2\theta}{|\mathbf{r}_i-\mathbf{r}_j|^3}$ the dipole-dipole potential, being μ_0 the magnetic permeability in vacuum, μ the magnetic dipole moment and θ the angle between the vector distance between dipoles and the polarization direction, which we choose as the x-axis, and $\varepsilon_{dd} = \mu_0\mu^2/(3g) = a_{dd}/a$ the ratio between the strength of the dipolar and the contact interaction, eventually written in terms of the dipolar length a_{dd} and the scattering length a.

In the case in which the condensate wave function is real (e.g., in absence of vortices, solitons...) the problem can be recast in a more convenient form. In fact, taking the sum and the difference between the two equations, one easily finds

$$(\hat{H}-\mu)(\hat{H}-\mu+2\hat{X})(u+v) = (\hbar\omega)^2(u+v)$$
$$(\hat{H}-\mu+2\hat{X})(\hat{H}-\mu)(u-v) = (\hbar\omega)^2(u-v)$$

Now, both equations allow to find the (square) of the energy of the Bogolyubov modes, but solving a system of half the dimensionality of the original problem. This typically allows a great saving of computational time. The eigenvectors of the two problems correspond to $(u+v)$ and $(u-v)$ respectively, so that if one is interested in finding the Bogolyubov quasi-particle amplitudes u and v, one also needs to solve the second problem, and then set $u = 0.5\,((u+v)+(u-v))$ and $v = 0.5\,((u+v)-(u-v))$

This class solves the eigenvalue problem using the matrix-free routines provided as part of the package `arpack-ng` , which is distributed as a bundled package with UltraCold.

2.4.1.2 Program description

As always, we first create an input file containing our mesh and physical parameters, as well as other parameters determining the run-time behavior of the system. Such input file will be called `example-4.prm` and contain the following text

```
# Mesh parameters

xmax = 20.0 # Size of the mesh along the x-axis, in micrometers. The mesh will extend from -xmax to xmax
ymax = 10.0 # Size of the mesh along the y-axis, in micrometers. The mesh will extend from -ymax to ymax
zmax = 20.0 # Size of the mesh along the z-axis, in micrometers. The mesh will extend from -zmax to zmax

nx = 48  # Number of points along the x-axis
ny = 48  # Number of points along the y-axis
nz = 256 # Number of points along the z-axis

# Physical parameters

scattering length = 95.0
dipolar_length      = 132.0 # Dipolar length in units of the Bohr radius
number of particles = 40000  # Total number of atoms
atomic mass         = 164    # Atomic mass, in atomic mass units
omegax = 110 # Harmonic frequency along the x-axis, in units of (2pi)Hz
omegay = 90 # Harmonic frequency along the y-axis, in units of (2pi)Hz
omegaz = 30 # Harmonic frequency along the z-axis, in units of (2pi)Hz

# Run parameters for gradient descent

number of gradient descent steps = 200000 # maximum number of gradient descent steps
residual                         = 1.E-12 # Threshold on the norm of the residual
alpha                            = 1.E-3 # gradient descent step
beta                             = 0.9   # step for the heavy-ball acceleration method

# Run parameters for Bogolyubov equations

number of modes = 50
calculate eigenvectors = true
tolerance = 1.E-8
maximum number of arnoldi iterations = 10000
```

We read the input file as usual using the class `Tools::InputParser` , set harmonic units, create a mesh and set the initial wave function and external potential for the calculation of the stationary state of the system, on top of which we are going to calculate the elementary excitations. Using the parameters above, such ground state is going to be a supersolid state.

```
#include "UltraCold.hpp"
#include <random>
using namespace UltraCold;
int main()
{
    Tools::InputParser ip("example-4.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    double zmax = ip.retrieve_double("zmax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    const int nz = ip.retrieve_int("nz");
    double scattering_length         = ip.retrieve_double("scattering length");
    double dipolar_length            = ip.retrieve_double("dipolar_length");
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    double omegaz                    = ip.retrieve_double("omegaz");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const int number_of_modes = ip.retrieve_int("number of modes");
    const int maximum_number_arnoldi_iterations = ip.retrieve_int("maximum number of arnoldi iterations");
    const double tolerance = ip.retrieve_double("tolerance");
    const bool calculate_eigenvectors = ip.retrieve_bool("calculate eigenvectors");
    const double hbar        = 0.6347*1.E5;
```

```
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    omegaz *= TWOPI;
    const double omega_ho = std::cbrt(omegax*omegay*omegaz);
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    omegaz = omegaz/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    scattering_length *= bohr_radius/a_ho;
    dipolar_length *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    zmax = zmax/a_ho;
    double dx = 2 * xmax / nx;
    double dy = 2 * ymax / ny;
    double dz = 2 * zmax / nz;
    Vector<double> x(nx), y(ny), z(nz), kx(nx), ky(ny), kz(nz);
    for (int i = 0; i < nx; ++i) x[i] = -xmax + i * dx;
    for (int i = 0; i < ny; ++i) y[i] = -ymax + i * dy;
    for (int i = 0; i < nz; ++i) z[i] = -zmax + i * dz;
    create_mesh_in_Fourier_space(x, y, z, kx, ky, kz);
    Vector<std::complex<double>> psi(nx, ny, nz);
    Vector<double> Vext(nx, ny, nz);
    std::default_random_engine generator;
    std::uniform_real_distribution<double> distribution(0,1);
    for (int i = 0; i < nx; ++i)
        for (int j = 0; j < ny; ++j)
            for (int k = 0; k < nz; ++k)
            {
                double random_number = distribution(generator);
                psi(i,j,k)  = (1.0+0.1*random_number)*
                              std::exp(-0.1*(pow(x(i),2) +
                                             pow(y(j),2) +
                                             pow(z(k),2)) );
                Vext(i,j,k) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                    std::pow(omegay,2)*pow(y(j),2) +
                                    std::pow(omegaz,2)*pow(z(k),2) );
            }
    double norm = 0.0;
    for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
    norm *= (dx * dy * dz);
    for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles / norm);
    UltraCold::RealSpaceOutput::DataOut psi_out;
    psi_out.set_output_name("initial_wave_function");
    psi_out.write_slice2d_vtk(x,y,psi,"xy","initial_wave_function");
```

So, we calculate the ground state of the system using the class `GPSolvers::DipolarGPSolver`

```
GPSolvers::DipolarGPSolver dipolar_gp_solver(x,
                                             y,
                                             z,
                                             psi,
                                             Vext,
                                             scattering_length,
                                             dipolar_length);
double chemical_potential;
std::tie(psi, chemical_potential) = dipolar_gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                           residual,
                                                                           alpha,
                                                                           beta,
                                                                           std::cout);
psi_out.set_output_name("ground_state_wave_function");
psi_out.write_vtk(x,y,z,psi,"ground_state_wave_function");
```

We can now plug the calculated ground-state solution ψ_0 to the class UltraCold::BogolyubovSolvers::TrappedDipolarBogolyubovSolver which will calculate for us the energies of the elementary excitations of the system as well as the Bogolyubov amplitudes u and v. Since we are considering a simple solution of the GPe, without any topological defect like solitons or vortices, the ground-state wave function, despite being defined as a complex Vector, will have only a non-zero real part. We can thus simplify the Bogolyubov equations solving only eigen-problems of halved dimensionality. The class `UltraCold::BogolyubovSolvers::TrappedDipolarBogolyubovSolver` will do this automatically for us, provided that we feed a real Vector, representing the ground-state wave function, to its constructor. We thus first copy the calculated ground-state wave-function into a real-valued Vector

```
Vector<double> psi_real(nx,ny,nz);
for (int i = 0; i < nx * ny * nz; ++i)
    psi_real[i] = psi[i].real();
```

then initialize the data structures that will contain the solutions of the Bogolyubov equations

```
std::vector<std::complex<double» eigenvalues(number_of_modes);
std::vector<Vector<std::complex<double»> u(number_of_modes),v(number_of_modes);
```

and, finally, create our solver class and run the solver

```
BogolyubovSolvers::TrappedDipolarBogolyubovSolver dipolar_bogolyubov_solver(x,
                                                                            y,
                                                                            z,
                                                                            psi_real,
                                                                            Vext,
                                                                            scattering_length,
                                                                            dipolar_length,
                                                                            chemical_potential,
                                                                            number_of_modes,
                                                                            tolerance,

       maximum_number_arnoldi_iterations,
                                                                            calculate_eigenvectors);
std::tie(eigenvalues,u,v) = dipolar_bogolyubov_solver.run();
```

In the context of Bogolyubov theory, several interesting properties can be extracted from the knowledge of u and v. For example, one can see the density and phase fluctuations associated with each eigen-mode, by looking, respectively, at the quantities

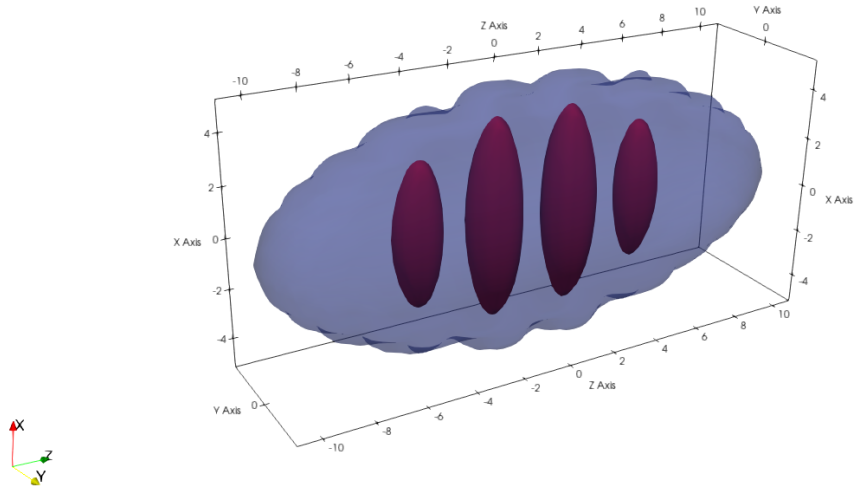
$$\delta n(\mathbf{r}) = (u(\mathbf{r}) + v(\mathbf{r}))\psi_0(\mathbf{r})$$
$$\delta \phi(\mathbf{r}) = (u(\mathbf{r}) - v(\mathbf{r}))/\psi_0(\mathbf{r})$$

This is exactly what we calculate and output into some .vtk files with the last lines of the example. Notice that we also print to the screen the expected frequencies of the center-of-mass oscillations of the system, which in the case of harmonic trapping coincide with the harmonic frequencies of the trap (they will, of course, be printed in units of their geometric average). This is a useful test of the accuracy of the calculation. If the calculations were accurate, one must find such three frequencies in the calculated energy spectrum.

```
    RealSpaceOutput::DataOut                    output_fluctuations;
    std::vector<Vector< std::complex<double> > density_fluctuations(number_of_modes);
    std::vector<Vector< std::complex<double> > phase_fluctuations(number_of_modes);
    std::cout << "----------------------------------"     << std::endl;
    std::cout << "Expected dipole mode frequencies: "     << std::endl;
    std::cout << omegax << " " << omegay << " " << omegaz << std::endl;
    std::cout << "----------------------------------"     << std::endl;
    for (int i = 0; i < number_of_modes; ++i)
    {
        std::cout << eigenvalues[i].real() << " " << eigenvalues[i].imag() << std::endl;
        if(calculate_eigenvectors)
        {
                density_fluctuations[i].reinit(nx,ny,nz);
                phase_fluctuations[i].reinit(nx,ny,nz);
                for (int j = 0; j < nx*ny*nz; ++j)
                {
                        density_fluctuations[i](j) = (u[i](j) + v[i](j)) * psi_real(j);
                        phase_fluctuations[i](j)   = (u[i](j) - v[i](j)) / psi_real(j);
                }
                output_fluctuations.set_output_name("density_fluctuations_mode_" + std::to_string(i));
                output_fluctuations.write_vtk(x,y,z,density_fluctuations[i], "density_fluctuations");
                output_fluctuations.set_output_name("phase_fluctuations_mode_" + std::to_string(i));
                output_fluctuations.write_vtk(x,y,z,phase_fluctuations[i], "phase_fluctuations");
        }
    }
    return 0;
}
```

2.4.1.3 Results

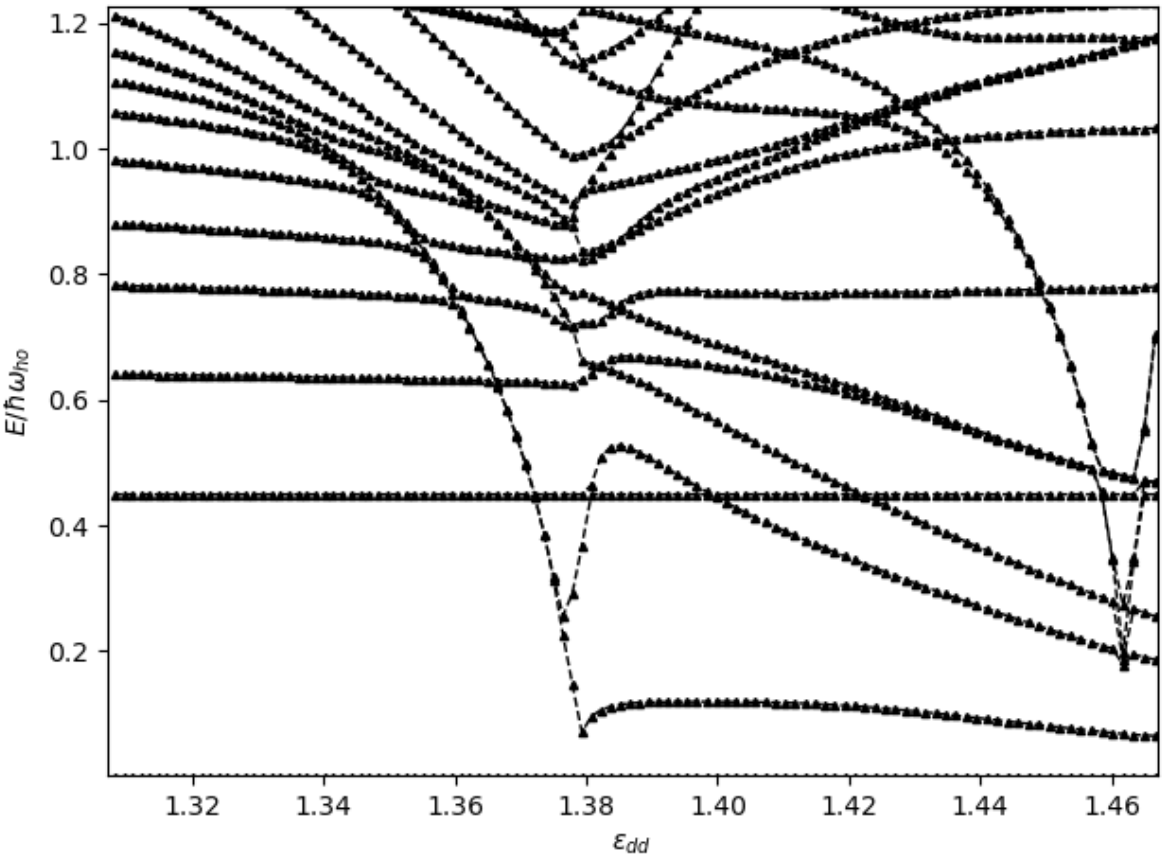
It is first interesting to get a look at the calculated ground-state wave function. As said, with the parameters used here, it corresponds to a supersolid state, as we can see from the three-dimensional density contours



The output of the program related to the (first) calculated eigenvalues, corresponding to the energies of the Bogolyubov modes in units of $\hbar\omega_{ho}$, is the following:

```
--------------------------------------
Expected dipole-mode frequencies:
1.6487 1.34893 0.449644
--------------------------------------
7.35959e-06 0
0.0990846 0
0.290267 0
0.388278 0
0.449645 0
0.564145 0
0.569203 0
0.772583 0
0.993786 0
1.01476 0
1.01743 0
1.08948 0
1.09409 0
1.18436 0
1.2107 0
1.30721 0
1.31027 0
1.34772 0
1.5384 0
1.54054 0
1.58407 0
1.60403 0
1.61188 0
1.62087 0
1.66431 0
1.66592 0
1.67126 0
1.6948 0
```

Notice that we catch well the two lowest dipole frequencies, and a little less well the higher energy one. This is not too much surprising, however, since at high energies even the dipole mode can be slightly affected by the other modes of comparable energy. It is also interesting to have a look at how the excitation spectrum changes when we tune the dipolar parameter $\varepsilon_{dd} = \mu_0\mu^2/(3g) = a_{dd}/a$. The results look like the following



Note

Since Bogolyubov calculations in three space dimensions are numerically very demanding, most of the calculations required to obtain the results presented in this example have been done on the High Performance Computing cluster Galileo100 of the Italian supercomputing consortium CINECA .

2.4.1.4 Possible extensions

One can use the calculated Bogolyubov amplitudes u and v, as well as the calculated density and phase fluctuations, to study for example the dynamic structure factor of the system, which describes the response of the system to small density probes, or the "character" of the modes, i.e. if a certain mode has mainly a density or a phase character. These kind of studies have led, in recent years, to several interesting publications, see for example Nature volume 574, pages 382-385 (2019) , Nature volume 574, pages 386-389 (2019) , and Phys. Rev. Lett. 123, 050402 (2019)

2.4.1.5 The plain program

```
/*-----------------------------------------------------------------------------
 *
 *    This file is part of the UltraCold project.
 *
 *    UltraCold is free software: you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation, either version 3 of the License, or
 *    any later version.
 *    UltraCold is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *    You should have received a copy of the GNU General Public License
 *    along with UltraCold.  If not, see <https://www.gnu.org/licenses/>.
 *
 *-----------------------------------------------------------------------------*/
#include "UltraCold.hpp"
#include <random>
using namespace UltraCold;
int main()
{
    Tools::InputParser ip("example-4.prm");
    ip.read_input_file();
    double xmax = ip.retrieve_double("xmax");
    double ymax = ip.retrieve_double("ymax");
    double zmax = ip.retrieve_double("zmax");
    const int nx = ip.retrieve_int("nx");
    const int ny = ip.retrieve_int("ny");
    const int nz = ip.retrieve_int("nz");
    double scattering_length         = ip.retrieve_double("scattering length");
    double dipolar_length            = ip.retrieve_double("dipolar_length");
    const int    number_of_particles = ip.retrieve_int("number of particles");
    const double atomic_mass         = ip.retrieve_double("atomic mass");
    double omegax                    = ip.retrieve_double("omegax");
    double omegay                    = ip.retrieve_double("omegay");
    double omegaz                    = ip.retrieve_double("omegaz");
    const int    number_of_gradient_descent_steps = ip.retrieve_int("number of gradient descent steps");
    const double residual                         = ip.retrieve_double("residual");
    const double alpha                            = ip.retrieve_double("alpha");
    const double beta                             = ip.retrieve_double("beta");
    const int number_of_modes = ip.retrieve_int("number of modes");
    const int maximum_number_arnoldi_iterations = ip.retrieve_int("maximum number of arnoldi iterations");
    const double tolerance = ip.retrieve_double("tolerance");
    const bool calculate_eigenvectors = ip.retrieve_bool("calculate eigenvectors");
    const double hbar        = 0.6347*1.E5;
    const double bohr_radius = 5.292E-5;
    omegax *= TWOPI;
    omegay *= TWOPI;
    omegaz *= TWOPI;
    const double omega_ho = std::cbrt(omegax*omegay*omegaz);
    omegax = omegax/omega_ho;
    omegay = omegay/omega_ho;
    omegaz = omegaz/omega_ho;
    const double a_ho = std::sqrt(hbar/(atomic_mass*omega_ho));
    scattering_length *= bohr_radius/a_ho;
    dipolar_length *= bohr_radius/a_ho;
    xmax = xmax/a_ho;
    ymax = ymax/a_ho;
    zmax = zmax/a_ho;
    double dx = 2 * xmax / nx;
    double dy = 2 * ymax / ny;
    double dz = 2 * zmax / nz;
    Vector<double> x(nx), y(ny), z(nz), kx(nx), ky(ny), kz(nz);
    for (int i = 0; i < nx; ++i) x[i] = -xmax + i * dx;
    for (int i = 0; i < ny; ++i) y[i] = -ymax + i * dy;
    for (int i = 0; i < nz; ++i) z[i] = -zmax + i * dz;
    create_mesh_in_Fourier_space(x, y, z, kx, ky, kz);
    Vector<std::complex<double>> psi(nx, ny, nz);
    Vector<double> Vext(nx, ny, nz);
    std::default_random_engine generator;
    std::uniform_real_distribution<double> distribution(0,1);
    for (int i = 0; i < nx; ++i)
        for (int j = 0; j < ny; ++j)
            for (int k = 0; k < nz; ++k)
            {
                double random_number = distribution(generator);
                psi(i,j,k)  = (1.0+0.1*random_number)*
                              std::exp(-0.1*(pow(x(i),2) +
                                             pow(y(j),2) +
                                             pow(z(k),2)) );
                Vext(i,j,k) = 0.5*( std::pow(omegax,2)*pow(x(i),2) +
                                    std::pow(omegay,2)*pow(y(j),2) +
                                    std::pow(omegaz,2)*pow(z(k),2) );
            }
    double norm = 0.0;
```

```
    for (size_t i = 0; i < psi.size(); ++i) norm += std::norm(psi[i]);
    norm *= (dx * dy * dz);
    for (size_t i = 0; i < psi.size(); ++i) psi[i] *= std::sqrt(number_of_particles / norm);
    UltraCold::RealSpaceOutput::DataOut psi_out;
    psi_out.set_output_name("initial_wave_function");
    psi_out.write_slice2d_vtk(x,y,psi,"xy","initial_wave_function");
    GPSolvers::DipolarGPSolver dipolar_gp_solver(x,
                                                 y,
                                                 z,
                                                 psi,
                                                 Vext,
                                                 scattering_length,
                                                 dipolar_length);
    double chemical_potential;
    std::tie(psi, chemical_potential) =
       dipolar_gp_solver.run_gradient_descent(number_of_gradient_descent_steps,
                                                                                residual,
                                                                                alpha,
                                                                                beta,
                                                                                std::cout);
    psi_out.set_output_name("ground_state_wave_function");
    psi_out.write_vtk(x,y,z,psi,"ground_state_wave_function");
    Vector<double> psi_real(nx,ny,nz);
    for (int i = 0; i < nx * ny * nz; ++i)
        psi_real[i] = psi[i].real();
    std::vector<std::complex<double» eigenvalues(number_of_modes);
    std::vector<Vector<std::complex<double»> u(number_of_modes),v(number_of_modes);
    BogolyubovSolvers::TrappedDipolarBogolyubovSolver dipolar_bogolyubov_solver(x,
                                                                                y,
                                                                                z,
                                                                                psi_real,
                                                                                Vext,
                                                                                scattering_length,
                                                                                dipolar_length,
                                                                                chemical_potential,
                                                                                number_of_modes,
                                                                                tolerance,

       maximum_number_arnoldi_iterations,
                                                                                calculate_eigenvectors);
    std::tie(eigenvalues,u,v) = dipolar_bogolyubov_solver.run();
    RealSpaceOutput::DataOut                    output_fluctuations;
    std::vector<Vector< std::complex<double> » density_fluctuations(number_of_modes);
    std::vector<Vector< std::complex<double> » phase_fluctuations(number_of_modes);
    std::cout « "----------------------------------"     « std::endl;
    std::cout « "Expected dipole mode frequencies: "     « std::endl;
    std::cout « omegax « " " « omegay « " " « omegaz « std::endl;
    std::cout « "----------------------------------"     « std::endl;
    for (int i = 0; i < number_of_modes; ++i)
    {
        std::cout « eigenvalues[i].real() « " " « eigenvalues[i].imag() « std::endl;
        if(calculate_eigenvectors)
        {
                density_fluctuations[i].reinit(nx,ny,nz);
                phase_fluctuations[i].reinit(nx,ny,nz);
                for (int j = 0; j < nx*ny*nz; ++j)
                {
                        density_fluctuations[i](j) = (u[i](j) + v[i](j)) * psi_real(j);
                        phase_fluctuations[i](j)   = (u[i](j) - v[i](j)) / psi_real(j);
                }
                output_fluctuations.set_output_name("density_fluctuations_mode_" + std::to_string(i));
                output_fluctuations.write_vtk(x,y,z,density_fluctuations[i], "density_fluctuations");
                output_fluctuations.set_output_name("phase_fluctuations_mode_" + std::to_string(i));
                output_fluctuations.write_vtk(x,y,z,phase_fluctuations[i], "phase_fluctuations");
        }
    }
    return 0;
}
```

Chapter 3

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

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

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

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

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see https://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read https://www.gnu.org/licenses/why-not-lgpl.html.

Chapter 4

Namespace Index

4.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 5

Hierarchical Index

5.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 6

Class Index

6.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 7

Namespace Documentation

7.1 UltraCold Namespace Reference

All the classes and functions necessary to work with UltraCold.

Namespaces

- namespace BogolyubovSolvers

  Solver classes for several flavours of Bogolyubov equations.
- namespace FourierSpaceOutput

  Classes and functions to output a data Vector in Fourier space.
- namespace GPSolvers

  Solver classes for various flavors of Gross-Pitaveskii equations.
- namespace RealSpaceOutput

  Classes and functions to output a data Vector in real space.
- namespace Tools

  Classes and functions of general utility, from input parsers to hardware inspectors.

Classes

- class Vector

  A class that represents arrays of numerical elements.

Functions

- void create_mesh_in_Fourier_space (Vector< double > &x, Vector< double > &kx)

  Creation of a mesh in Fourier space for a one-dimensional problem.
- void create_mesh_in_Fourier_space (Vector< double > &x, Vector< double > &y, Vector< double > &kx, Vector< double > &ky)

  Creation of a mesh in Fourier space for a two-dimensional problem.
- void create_mesh_in_Fourier_space (Vector< double > &x, Vector< double > &y, Vector< double > &z, Vector< double > &kx, Vector< double > &ky, Vector< double > &kz)

  Creation of a mesh in Fourier space for a three-dimensional problem.

7.1.1 Detailed Description

All the classes and functions necessary to work with UltraCold.

7.1.2 Function Documentation

7.1.2.1 create_mesh_in_Fourier_space() [1/3]

```
void UltraCold::create_mesh_in_Fourier_space (
            Vector< double > & x,
            Vector< double > & kx )
```

Creation of a mesh in Fourier space for a one-dimensional problem.

Function to create a mesh in Fourier space.

Parameters

x	*Vector<double>* representing the x-axis of a cartesian reference frame
kx	*Vector<double>* representing the kx-axis of the corresponding Fourier space

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

This function allows to generate a mesh in Fourier space, starting from a corresponding mesh in real, cartesian space. It is also useful to plot the Fourier transform of a Vector.
The function is used as follows

```
create_mesh_in_Fourier_space(x,kx);          // for a 1D problem
create_mesh_in_Fourier_space(x,y,kx,ky);     // for a 2D problem
create_mesh_in_Fourier_space(x,y,z,kx,ky,kz) // for a 3D problem
```

This will take the three Vectors x,y and z, representing a mesh in real space, and use them to generate the corresponding mesh in Fourier space.

7.1.2.2 create_mesh_in_Fourier_space() [2/3]

```
void UltraCold::create_mesh_in_Fourier_space (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & kx,
            Vector< double > & ky )
```

Creation of a mesh in Fourier space for a two-dimensional problem.

Parameters

x	*Vector<double>* representing the x-axis of a cartesian reference frame
y	*Vector<double>* representing the y-axis of a cartesian reference frame
kx	*Vector<double>* representing the kx-axis of the corresponding Fourier space
ky	*Vector<double>* representing the ky-axis of the corresponding Fourier space

7.1.2.3 create_mesh_in_Fourier_space() [3/3]

```
void UltraCold::create_mesh_in_Fourier_space (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & z,
            Vector< double > & kx,
            Vector< double > & ky,
            Vector< double > & kz )
```

Creation of a mesh in Fourier space for a three-dimensional problem.

Parameters

x	*Vector<double>* representing the x-axis of a cartesian reference frame
y	*Vector<double>* representing the y-axis of a cartesian reference frame
z	*Vector<double>* representing the z-axis of a cartesian reference frame
kx	*Vector<double>* representing the kx-axis of the corresponding Fourier space
ky	*Vector<double>* representing the ky-axis of the corresponding Fourier space
kz	*Vector<double>* representing the kz-axis of the corresponding Fourier space

7.2 UltraCold::BogolyubovSolvers Namespace Reference

Solver classes for several flavours of Bogolyubov equations.

Classes

- class TrappedBogolyubovSolver

  Class to solve the Bogolyubov equations for a trapped Bose gas.
- class TrappedDipolarBogolyubovSolver

  *Class to solve the Bogolyubov equations for a trapped **dipolar** Bose gas.*

7.2.1 Detailed Description

Solver classes for several flavours of Bogolyubov equations.

While the GPSolver class and related classes allows to study the ground state and the dynamics of Bose-Einstein condensates, the elementary excitations on top of a certain state can be studied by solving the so-called *Bogolyubov equations*. The idea is to consider a certain configuration, described a condensate wave-function ψ_0, and with chemical potential μ, and to study small oscillations on top of it by searching for solutions of the Gross-Pitaevskii equation of the form

$$\psi(\mathbf{r},t) = e^{-i\frac{\mu}{\hbar}t}\left[\psi_0(\mathbf{r}) + \sum_{n=0}^{\infty}\left(u_n(\mathbf{r})e^{-i\omega_n t} + v_n^*(\mathbf{r})e^{i\omega_n t}\right)\right]$$

Keeping only terms linear the functions u and v, one finds that the *quasi-particle amplitudes* u_n and v_n and their energies $\hbar\omega_n$ are given by the solutions of a certain eigenvalue problem. The form depends on the "flavour" of Gross-Pitaevskii equation one is considering (ordinary, dipolar, mixtures...) and is described in the documentation of the specific solver classes.

7.3 UltraCold::FourierSpaceOutput Namespace Reference

Classes and functions to output a data Vector in Fourier space.

Classes

- class DataOut

  A class to output a data Vector in Fourier space.

7.3.1 Detailed Description

Classes and functions to output a data Vector in Fourier space.

7.4 UltraCold::GPSolvers Namespace Reference

Solver classes for various flavors of Gross-Pitaveskii equations.

Classes

- class DipolarGPSolver

  Class to solve the Gross-Pitaevskii equation for a dipolar Bose gas with the Lee-Huang-Yang correction.
- class GPSolver

  Class to solve the Gross-Pitaevskii equation.

7.4.1 Detailed Description

Solver classes for various flavors of Gross-Pitaveskii equations.

Ultra-cold bosonic systems are very often described in terms of the so-called Gross-Pitaevskii equation , which is a non-linear Schrodinger equation arising from a mean-field description of the system, valid when the particles are very weakly interacting and the temperature is close to the absolute zero. In these conditions, an ultra-cold gas of weakly interacting bosonic atoms is described by a complex order parameter ψ, whose square modulus gives the local density of atoms, and satisfying the following Gross-Pitaevskii equation

$$i\hbar\frac{\partial\psi}{\partial t} = \left[\frac{-\hbar^2\nabla^2}{2m} + V_{ext}(\mathbf{r}) + g|\psi|^2\right]\psi$$

Here V_{ext} is some external trapping potential, and g is related to the s-wave scattering length a by $g = \frac{4\pi\hbar^2 a}{m}$. This model is valid as long as the two-body interaction between the atoms can be modeled by a contact interaction of the form $V(\mathbf{r}-\mathbf{r}') = g\delta(\mathbf{r}-\mathbf{r}')$. Other models (including, for example, a dipole-dipole interaction and the effects of quantum fluctuations) have a similar form, as it is discussed in other solver classes belonging to this namespace.
There are two kinds of information we can derive from the solution of the Gross-Pitaevskii equation:

- *Ground-state properties*: these are obtained by searching for stationary solutions of the form $\psi(\mathbf{r},t) = \psi_0(\mathbf{r})e^{-i\mu t/\hbar}$, obtaining the stationary Gross-Pitaevskii equation

$$\mu\psi_0 = \left[\frac{-\hbar^2\nabla^2}{2m} + V_{ext}(\mathbf{r}) + g|\psi_0|^2\right]\psi_0$$

Solving this for the smallest eigenvalue μ, which represents the chemical potential, gives access to the ground-state configuration of the system.

- *Dynamics*: solving the Gross-Pitaevskii equation for appropriate initial conditions allows to simulate the dynamical behavior of the system and to compare the results of the model with experiments.

In order to solve these equations, there are several possibilities.
For what concern the stationary Gross-Pitaevskii equation, one possibility comes from noticing that such equation can be obtained from a constrained minimization formulation of the problem, in particular by requiring that the ground-state order parameter of the system is the exact minimizer of the mean-field energy functional

$$E[\psi] = \int d\mathbf{r}\left[\psi^*(\mathbf{r})\left(\frac{-\hbar^2\nabla^2}{2m} + V_{ext}(\mathbf{r})\right)\psi(\mathbf{r})\right] + \frac{g}{2}\int d\mathbf{r}|\psi(\mathbf{r})|^4$$

under the constraint of a fixed number of particles $\int d\mathbf{r}|\psi(\mathbf{r})|^2 = N$. This allows also to introduce the chemical potential μ as the Lagrange multiplier fixing the total number of particles. One can hence find the ground state order parameter and chemical potential using, for example, a gradient descent method, as implemented in the class GPSolver (see the related documentation).
For what concern instead the full Gross-Pitaevskii equation, this is solved using a classical operator-splitting method. See again the full documentation of the GPSolver class for more details.

7.5 UltraCold::RealSpaceOutput Namespace Reference

Classes and functions to output a data Vector in real space.

Classes

- class DataOut

  A class to output a data Vector in real space.

7.5.1 Detailed Description

Classes and functions to output a data Vector in real space.

7.6 UltraCold::Tools Namespace Reference

Classes and functions of general utility, from input parsers to hardware inspectors.

Classes

- class HardwareInspector

  Class to detect hardware capabilities.
- class InputParser

  Class to read input parameters from files.

7.6.1 Detailed Description

Classes and functions of general utility, from input parsers to hardware inspectors.

Chapter 8

Class Documentation

8.1 UltraCold::FourierSpaceOutput::DataOut Class Reference

A class to output a data Vector in Fourier space.

`#include <data_out.hpp>`

Public Member Functions

- void **set_output_name** (const std::string &output_file_name)

  Set the name for the output data file, input as an std::string.
- void **set_output_name** (const char *output_file_name)

  Set the name for the output data file, input as simple text.
- void write_csv (Vector< double > &x_axis, Vector< std::complex< double > > &output_vector)

  Write an output data file in .csv format, for complex 1D output in Fourier space.
- void write_csv (Vector< double > &x_axis, Vector< double > &y_axis, Vector< std::complex< double > > &output_vector)

  Write an output data file in .csv format, for complex 2D output in Fourier space.
- void write_slice1d_csv (Vector< double > &axis, Vector< std::complex< double > > &output_vector, const char *axis_name)

  Write an output data file in .csv format, for 1D slice of complex 2D or 3D Vector in Fourier space.
- void write_slice2d_csv (Vector< double > &x_axis, Vector< double > &y_axis, Vector< std::complex< double > > &output_vector, const char *axis_name)

  Write an output data file in .csv format, for 2D slice of complex 3D Vector in Fourier space.

8.1.1 Detailed Description

A class to output a data Vector in Fourier space.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

This class allows to specifically write output data files in the following formats

- .csv

for a Vector living in Fourier space, for example obtained from a Fourier transform calculated using the class DFt↩ Calculator. The usage is exactly the same as the corresponding class for output of data files in real space, with the important difference that, in this case, the output Vector is always of type complex.

Note

For proper output, the axis in momentum space must have been generated by a call to the function generate↩ _mesh_in_Fourier_space()

8.1.2 Member Function Documentation

8.1.2.1 write_csv() [1/2]

```
void UltraCold::FourierSpaceOutput::DataOut::write_csv (
            Vector< double > & x,
            Vector< double > & y,
            Vector< std::complex< double > > & v )
```

Write an output data file in .csv format, for complex 2D output in Fourier space.

Parameters

kx	*Vector<double>* the first Vector for the output. This is considered as the kx-axis for the plot. For proper output, it must have been generated with a call to MKLWrappers::generate_mesh_in_Fourier_space.
ky	*Vector<double>* the second Vector for the output. This is considered as the kx-axis for the plot. For proper output, it must have been generated with a call to MKLWrappers::generate_mesh_in_Fourier_space.
v	*Vector<double>* the third Vector for the output. This is considered as the data to be plotted on the z-axis of the plot.

Notice that we have two possible cases: either `v` comes from a real-to-complex transform, in which case its size along the last extent will be one-half (plus one) the size of `kx`, or it comes from a complex-to-complex transform, in which case they will be equal. In each case, the extent along the second direction of `v` must be the same as the one of `ky`. The output format will be similar to the one of the corresponding member functions of the class RealSpaceOutput::DataOut, and can be plotted using, for example, `gnuplot` in the same way. Check the corresponding documentation for RealSpaceOutput::DataOut for more information.

8.1.2.2 write_csv() [2/2]

```
void UltraCold::FourierSpaceOutput::DataOut::write_csv (
```

```
            Vector< double > & x,
            Vector< std::complex< double > > & v )
```

Write an output data file in .csv format, for complex 1D output in Fourier space.

Parameters

kx	*Vector<double>* the first Vector for the output. This is considered as the kx-axis for the plot. For proper output, it must have been generated with a call to MKLWrappers::generate_mesh_in_Fourier_space.
v	*Vector<std::complex<double>>* the second Vector for the output. This is considered as the data to be plotted on the y-axis of the plot.

Notice that we have two possible cases: either `v` comes from a real-to-complex transform, in which case its size will be one-half (plus one) the size of `kx`, or it comes from a complex-to-complex transform, in which case the the two vectors will have the same size. In each case, the output format will be similar to the one of the corresponding member functions of the class RealSpaceOutput::DataOut, and can be plotted using, for example, `gnuplot` in the same way. Check the corresponding documentation for RealSpaceOutput::DataOut for more information.

8.1.2.3 write_slice1d_csv()

```
void UltraCold::FourierSpaceOutput::DataOut::write_slice1d_csv (
            Vector< double > & x,
            Vector< std::complex< double > > & v,
            const char * axis )
```

Write an output data file in .csv format, for 1D slice of complex 2D or 3D Vector in Fourier space.

Parameters

kx	*Vector<double>* the first Vector for the output. This is considered as the kx-axis for the plot. For proper output, it must have been generated with a call to MKLWrappers::generate_mesh_in_Fourier_space.
v	*Vector<std::complex<double>>* the second Vector for the output. One of its cuts is considered as the y-axis of the plot.
axis	the axis along which the cut is taken. This can either be `"kx"`, `"ky"`, or `"kz"`. For a 2D Vector `v`, if `axis="kx"` a cut of the 2D Vector `v` will be taken along the ky=0 axis (and vice-versa). For a 3D Vector `v`, if `axis="kx"`, a cut along the intersection of the two planes kz=0 and ky=0 will be taken, ans similarly if `axis="ky"` or `axis="kz"`.

Notice that we have two possible cases: either `v` comes from a real-to-complex transform, in which case its size along the last extent will be one-half (plus one) the size of `ky` (or `kz`), or it comes from a complex-to-complex transform, in which case they will be equal. The output format will be similar to the one of the corresponding member functions of the class RealSpaceOutput::DataOut, and can be plotted using, for example, `gnuplot` in the same way. Check the corresponding documentation for RealSpaceOutput::DataOut for more information.

Warning

This function performs only a few range checking, hence be careful in passing consistent vectors as input. If the extents of the Vectors provided are not consistent, a segmentation fault may arise.

8.1.2.4 write_slice2d_csv()

```
void UltraCold::FourierSpaceOutput::DataOut::write_slice2d_csv (
            Vector< double > & x,
            Vector< double > & y,
            Vector< std::complex< double > > & v,
            const char * plane )
```

Write an output data file in .csv format, for 2D slice of complex 3D Vector in Fourier space.

Parameters

kx	*Vector<double>* the first Vector for the output. This is considered as the kx-axis for the plot. For proper output, it must have been generated with a call to MKLWrappers::generate_mesh_in_Fourier_space.
ky	*Vector<double>* the second Vector for the output. This is considered as the ky-axis for the plot. For proper output, it must have been generated with a call to MKLWrappers::generate_mesh_in_Fourier_space.
v	*Vector<double>* the third Vector for the output. One of its cuts is considered as the z-axis of the plot.
plane	the plane along which the cut is taken. This can either be `"kxy"`, `"kyz"`, or `"kxz"`. If `plane="kxy"`, a slice along the kz=0 plane will be taken, and similarly if `axis="kyz"` or `axis="kxz"`.

Notice that we have two possible cases: either `v` comes from a real-to-complex transform, in which case its size along the last extent will be one-half (plus one) the size of `ky` (or `kz`), or it comes from a complex-to-complex transform, in which case they will be equal. The output format will be similar to the one of the corresponding member functions of the class RealSpaceOutput::DataOut, and can be plotted using, for example, `gnuplot` in the same way. Check the corresponding documentation for RealSpaceOutput::DataOut for more information.

Warning

This function performs only a few range checking, hence be carefull in passing consistent vectors as input. If the extents of the Vectors provided are not consistent, a segmentation fault may arise.

8.2 UltraCold::RealSpaceOutput::DataOut Class Reference

A class to output a data Vector in real space.

`#include <data_out.hpp>`

Public Member Functions

- void **set_output_name** (const std::string &output_file_name)

  Set the name for the output data file, input as an std::string.
- void **set_output_name** (const char *output_file_name)

  Set the name for the output data file, input as simple text.
- void write_csv (Vector< double > &x_axis, Vector< double > &real_output_vector)

  Write an output data file in .csv format, for real 1D output.
- void write_csv (Vector< double > &x_axis, Vector< std::complex< double > > &complex_output_vector)

  Write an output data file in .csv format, for complex 1D output.

- void write_csv (Vector< double > &x_axis, Vector< double > &y_axis, Vector< double > &real_output_vector)

  Write an output data file in .csv format, for real 2D output.
- void write_csv (Vector< double > &x_axis, Vector< double > &y_axis, Vector< std::complex< double > > &complex_output_vector)

  Write an output data file in .csv format, for complex 2D output.
- void write_slice1d_csv (Vector< double > &axis, Vector< double > &real_output_vector, const char *axis_name)

  Write an output data file in .csv format, for 1D slice of real 2D or 3D Vector.
- void write_slice1d_csv (Vector< double > &axis, Vector< std::complex< double > > &complex_output_vector, const char *axis_name)

  Write an output data file in .csv format, for 1D slice of complex 2D or 3D Vector.
- void write_slice2d_csv (Vector< double > &x_axis, Vector< double > &y_axis, Vector< double > &real_output_vector, const char *plane_name)

  Write an output data file in .csv format, for 2D slice of real 3D Vector.
- void write_slice2d_csv (Vector< double > &x_axis, Vector< double > &y_axis, Vector< std::complex< double > > &complex_output_vector, const char *plane_name)

  Write an output data file in .csv format, for 2D slice of complex 3D Vector.
- void stack1d_csv (double time, Vector< double > &x_axis, Vector< double > &real_output_vector)

  Stack a one-dimensional, time-varying real vector for a two-dimensional plot.
- void stack1d_csv (double time, Vector< double > &x_axis, Vector< std::complex< double > > &complex_output_vector)

  Stack a one-dimensional, time-varying complex vector for a two-dimensional plot.
- void write_vtk (Vector< double > &x_axis, Vector< double > &y_axis, Vector< double > &real_output_vector, const char *vector_name)

  Write an output data file in .vtk format, for real 2D output.
- void write_vtk (Vector< double > &x_axis, Vector< double > &y_axis, Vector< std::complex< double > > &complex_output_vector, const char *vector_name)

  Write an output data file in .vtk format, for complex 2D output.
- void write_vtk (Vector< double > &x_axis, Vector< double > &y_axis, Vector< double > &z_axis, Vector< double > &real_output_vector, const char *vector_name)

  Write an output data file in .vtk format, for real 3D output.
- void write_vtk (Vector< double > &x_axis, Vector< double > &y_axis, Vector< double > &z_axis, Vector< std::complex< double > > &complex_output_vector, const char *vector_name)

  Write an output data file in .vtk format, for complex 3D output.
- void write_slice2d_vtk (Vector< double > &x_axis, Vector< double > &y_axis, Vector< double > &real_output_vector, const char *vector_name, const char *plane)

  Write an output data file in .vtk format, for 2D slice of real 3D Vector.
- void write_slice2d_vtk (Vector< double > &x_axis, Vector< double > &y_axis, Vector< std::complex< double > > &complex_output_name, const char *vector_name, const char *plane)

  Write an output data file in .vtk format, for 2D slice of complex 3D Vector.

8.2.1 Detailed Description

A class to output a data Vector in real space.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

This class allows to write output datafiles in different formats, ready for plots with common visualization tools such as gnuplot, matplotlib, Paraview or Visit. The currently supported data output formats are:

- .csv
- .vtk

The class is used as follows:

```
DataOut data_output_writer();
data_output_writer.set_output_name("MyOutputFile");
data_output_writer.write_csv(axis, Vector); // Or other appropriate member function
```

The first line creates an object of type DataOut named data_output_writer .
The second line sets the output file name to "MyOutputFile".
The third line provides an example for writing a Vector in a simple .csv file, ready to be plotted for example with gnuplot.
See the member function documentation for details on the different ways to output your data.

8.2.2 Member Function Documentation

8.2.2.1 stack1d_csv() [1/2]

```
void UltraCold::RealSpaceOutput::DataOut::stack1d_csv (
            double time,
            Vector< double > & x,
            Vector< double > & v )
```

Stack a one-dimensional, time-varying real vector for a two-dimensional plot.

Parameters

time	*double* the current instant of time
x	*Vector<double>* space axis
v	*Vector<double>* the vector representing the data for output

The output file will be a .csv file with a structure resembling the one of the files generated with, for example, write_csv() for two-dimensional plots of real Vectors. However, here, the first column will contain time, the second space, and third the data to output. This function should be placed inside a time-loop, since it appends new data at the bottom of the file every time it is called.

8.2.2.2 stack1d_csv() [2/2]

```
void UltraCold::RealSpaceOutput::DataOut::stack1d_csv (
            double time,
```

```
            Vector< double > & x,
            Vector< std::complex< double > > & v )
```

Stack a one-dimensional, time-varying complex vector for a two-dimensional plot.

Parameters

time	*double* the current instant of time
x	*Vector<double>* space axis
v	*Vector<std::complex<double>>* the vector representing the data for output

The output file will be a .csv file with a structure resembling the one of the files generated with, for example, write_csv() for two-dimensional plots of complex Vectors. However, here, the first column will contain time, the second space, and third and fourth the data to output. This function should be placed inside a time-loop, since it appends new data at the bottom of the file every time it is called.

8.2.2.3 write_csv() [1/4]

```
void UltraCold::RealSpaceOutput::DataOut::write_csv (
            Vector< double > & x,
            Vector< double > & v )
```

Write an output data file in .csv format, for real 1D output.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
v	*Vector<double>* the second Vector for the output. This is considered as the data to be plotted on the y-axis of the plot.

The output file will have the format

```
x(0),v(0)
x(1),v(1)
...  ...
x(n),v(n)
```

where `n` is the leading dimension of the two Vectors (which of course must have the same size).
To visualize the data written in the file using, for example, `gnuplot`, open `gnuplot` in a terminal and type the commands

```
set datafile separator ",";
plot "filename.csv"
```

8.2.2.4 write_csv() [2/4]

```
void UltraCold::RealSpaceOutput::DataOut::write_csv (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & v )
```

Write an output data file in .csv format, for real 2D output.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
v	*Vector<double>* the third Vector for the output. This is considered as the data to be plotted on the z-axis of the plot.

The output file will have the format

```
x(0),y(0),v(0,0)
x(0),y(1),v(0,1)
...  ...
x(0),y(ny-1),v(0,ny-1)

x(1),y(0),v(1,0)
x(1),y(1),v(1,1)
...  ...  ...
...  ...  ...

x(nx-1),y(ny-1),v(nx-1,ny-1)
```

where `nx` and `ny` are the dimensions of the Vectors (which of course must be consistent).
To visualize the data written in the file using, for example, `gnuplot`, open `gnuplot` in a terminal and type the commands

```
set datafile separator ",";
set pm3d;
set palette;
unset surface;
set view map;
splot "filename.csv" u 1:2:3
```

8.2.2.5 write_csv() [3/4]

```
void UltraCold::RealSpaceOutput::DataOut::write_csv (
            Vector< double > & x,
            Vector< double > & y,
            Vector< std::complex< double > > & v )
```

Write an output data file in .csv format, for complex 2D output.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
v	*Vector<std::complex<double>>* the third Vector for the output. This is considered as the data to be plotted on the z-axis of the plot.

The output file will have the format

```
x(0),y(0),v'(0,0),v''(0,0)
x(0),y(1),v'(0,1),v''(0,1)
...  ...
x(0),y(ny-1),v'(0,ny-1),v''(0,ny-1)
```

```
x(1),y(0),v'(1,0),v''(1,0)
x(1),y(1),v'(1,1),v''(1,1)
...  ...  ...
...  ...  ...

x(nx-1),y(ny-1),v'(nx-1,ny-1),v''(nx-1,ny-1)
```

where nx and ny are the dimensions of the Vectors (which of course must be consistent), v' is the real part of the Vector v, and v" is its imaginary part.
To visualize the data written in the file using, for example, gnuplot, open gnuplot in a terminal and type the commands

```
set datafile separator ",";
set pm3d;
set palette;
unset surface;
set view map;
splot "filename.csv" u 1:2:3 # for real() part, for the imaginary u 1:2:4
```

8.2.2.6 write_csv() [4/4]

```
void UltraCold::RealSpaceOutput::DataOut::write_csv (
            Vector< double > & x,
            Vector< std::complex< double > > & v )
```

Write an output data file in .csv format, for complex 1D output.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
v	*Vector<std::complex<double>>* the second Vector for the output. This is considered as the data to be plotted on the y-axis of the plot.

The output file will have the format

```
x(0),v'(0),v''(0)
x(1),v'(1),v''(1)
...  ...
x(n),v'(n),v''(n)
```

where n is the leading dimension of the two Vectors (which of course must have the same size), v' is the real part of Vector v, and v" is its imaginary part.
To visualize the data written in the file using, for example, gnuplot, open gnuplot in a terminal and type the commands

```
set datafile separator ",";
plot "filename.csv" u 1:2 # for the real part, u 1:3 for the imaginary part
```

8.2.2.7 write_slice1d_csv() [1/2]

```
void UltraCold::RealSpaceOutput::DataOut::write_slice1d_csv (
            Vector< double > & x,
            Vector< double > & v,
            const char * axis )
```

Write an output data file in .csv format, for 1D slice of real 2D or 3D Vector.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
v	*Vector<double>* the second Vector for the output. One of its cuts is considered as the y-axis of the plot.
axis	the axis along which the cut is taken. This can either be "x", "y", or "z". For a 2D Vector v, if axis="x" a cut of the 2D Vector v will be taken along the y=0 axis, in the second along the x=0 axis. For a 3D Vector v, if axis="x", a cut along the intersection of the two planes z=0 and y=0 will be taken, ans similarly if axis="y" or axis="z".

The format of the output data file resembles the one of 1D plots.

8.2.2.8 write_slice1d_csv() [2/2]

```
void UltraCold::RealSpaceOutput::DataOut::write_slice1d_csv (
            Vector< double > & x,
            Vector< std::complex< double > > & v,
            const char * axis )
```

Write an output data file in .csv format, for 1D slice of complex 2D or 3D Vector.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
v	*Vector<double>* the second Vector for the output. One of its cuts is considered as the y-axis of the plot.
axis	the axis along which the cut is taken. This can either be "x", "y", or "z". For a 2D Vector v, if axis="x" a cut of the 2D Vector v will be taken along the y=0 axis, in the second along the x=0 axis. For a 3D Vector v, if axis="x", a cut along the intersection of the two planes z=0 and y=0 will be taken, ans similarly if axis="y" or axis="z".

The format of the output data file resembles the one of 1D plots.

8.2.2.9 write_slice2d_csv() [1/2]

```
void UltraCold::RealSpaceOutput::DataOut::write_slice2d_csv (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & v,
            const char * plane )
```

Write an output data file in .csv format, for 2D slice of real 3D Vector.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
v	*Vector<double>* the third Vector for the output. One of its cuts is considered as the z-axis of the plot.
plane	the plane along which the cut is taken. This can either be "xy", "yz", or "xz". If plane="xy", a slice along the z=0 plane will be taken, and similarly if axis="yz" or axis="xz".

The format of the output data file resembles the one of 2D plots.

8.2.2.10 write_slice2d_csv() [2/2]

```
void UltraCold::RealSpaceOutput::DataOut::write_slice2d_csv (
            Vector< double > & x,
            Vector< double > & y,
            Vector< std::complex< double > > & v,
            const char * plane )
```

Write an output data file in .csv format, for 2D slice of complex 3D Vector.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
v	*Vector<double>* the third Vector for the output. One of its cuts is considered as the z-axis of the plot.
plane	the plane along which the cut is taken. This can either be `"xy"`, `"yz"`, or `"xz"`. If `plane="xy"`, a slice along the z=0 plane will be taken, and similarly if `axis="yz"` or `axis="xz"`.

The format of the output data file resembles the one of 2D plots.

8.2.2.11 write_slice2d_vtk() [1/2]

```
void UltraCold::RealSpaceOutput::DataOut::write_slice2d_vtk (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & v,
            const char * plane,
            const char * output_vector_name )
```

Write an output data file in .vtk format, for 2D slice of real 3D Vector.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
v	*Vector<double>* the third Vector for the output. One of its cuts is considered as the z-axis of the plot.
plane	the plane along which the cut is taken. This can either be `"xy"`, `"yz"`, or `"xz"`. If `plane="xy"`, a slice along the z=0 plane will be taken, and similarly if `axis="yz"` or `axis="xz"`.
output_vector_name	*char* the name of the output

The output file will be in standard .vtk format for structured data points. It can be readily visualized using programs like `Paraview` or `Visit`.

8.2.2.12 write_slice2d_vtk() [2/2]

```
void UltraCold::RealSpaceOutput::DataOut::write_slice2d_vtk (
            Vector< double > & x,
            Vector< double > & y,
            Vector< std::complex< double > > & v,
            const char * plane,
            const char * output_vector_name )
```

Write an output data file in .vtk format, for 2D slice of complex 3D Vector.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
v	*Vector<std::complex<double>>* the third Vector for the output. One of its cuts is considered as the z-axis of the plot.
plane	the plane along which the cut is taken. This can either be `"xy"`, `"yz"`, or `"xz"`. If `plane="xy"`, a slice along the z=0 plane will be taken, and similarly if `axis="yz"` or `axis="xz"`.
output_vector_name	*char* the name of the output vector.

The output file will be in standard .vtk format for structured data points. It can be readily visualized using programs like Paraview or Visit.

8.2.2.13 write_vtk() [1/4]

```
void UltraCold::RealSpaceOutput::DataOut::write_vtk (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & v,
            const char * output_vector_name )
```

Write an output data file in .vtk format, for real 2D output.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
v	*Vector<double>* the third Vector for the output. This is considered as the data to be plotted on the z-axis of the plot.
output_vector_name	*char* the name of the output vector

The output file will be in standard .vtk format for structured data points. It can be readily visualized using programs like Paraview or Visit.

8.2.2.14 write_vtk() [2/4]

```
void UltraCold::RealSpaceOutput::DataOut::write_vtk (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & z,
            Vector< double > & v,
            const char * output_vector_name )
```

Write an output data file in .vtk format, for real 3D output.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
z	*Vector<double>* the third Vector for the output. This is considered as the z-axis for the plot.
v	*Vector<double>* the fourth Vector for the output.
output_vector_name	*char* the name of the output vector.

The output file will be in standard .vtk format for structured data points. It can be readily visualized using programs like Paraview or Visit.

8.2.2.15 write_vtk() [3/4]

```
void UltraCold::RealSpaceOutput::DataOut::write_vtk (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & z,
            Vector< std::complex< double > > & v,
            const char * output_vector_name )
```

Write an output data file in .vtk format, for complex 3D output.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
z	*Vector<double>* the third Vector for the output. This is considered as the z-axis for the plot.
v	*Vector<std::complex<double>>* the fourth Vector for the output.
output_vector_name	*char* the name of the output vector.

The output file will be in standard .vtk format for structured data points. It can be readily visualized using programs like Paraview or Visit.

8.2.2.16 write_vtk() [4/4]

```
void UltraCold::RealSpaceOutput::DataOut::write_vtk (
            Vector< double > & x,
            Vector< double > & y,
            Vector< std::complex< double > > & v,
            const char * output_vector_name )
```

Write an output data file in .vtk format, for complex 2D output.

Parameters

x	*Vector<double>* the first Vector for the output. This is considered as the x-axis for the plot.
y	*Vector<double>* the second Vector for the output. This is considered as the y-axis for the plot.
v	*Vector<std::complex<double>>* the third Vector for the output. This is considered as the data to be plotted on the z-axis of the plot.
output_vector_name	*char* the name of the output vector.

The output file will be in standard .vtk format for structured data points. It can be readily visualized using programs like Paraview or Visit.

8.3 UltraCold::MKLWrappers::DFtCalculator Class Reference

Class to calculate Fourier transforms using Intel's MKL DFT functions.

`#include <dft.hpp>`

Public Member Functions

- DFtCalculator (Vector< std::complex< double > > &forward_domain_vector, Vector< std::complex< double > > &backward_domain_vector)

  Constructor for complex-complex transforms.
- DFtCalculator (Vector< double > &forward_domain_vector, Vector< std::complex< double > > &backward_domain_vector)

  Constructor for real-complex transforms.
- ~**DFtCalculator** ()

  Destructor, free the descriptor handler.
- void **compute_forward** ()

  Calculate a forward transform.
- void compute_backward ()

  Calculate a backward transform.

8.3.1 Detailed Description

Class to calculate Fourier transforms using Intel's MKL DFT functions.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

The purpose of this class is to provide a simple way for calculating Fourier transforms using Intel's MKL Discrete Fourier Transform functions.
The class is used as follows

```
DFtCalculator dft(forward_domain_vector,backward_domain_vector);
dft.compute_forward(); // For a forward transform
// or
dft.compute_backward(); // for a backward transform
```

Here, `forward_domain_vector` is the real-space vector, while `backward_domain_vector` is the Fourier-space vector.

Note

The two Vectors must be different (out-of-place transform). Moreover, for performance and portability reasons, a `DFtCalculator` object must be associated with **each** forward-backward pair of vectors you want to use.

8.3.2 Constructor & Destructor Documentation

8.3.2.1 DFtCalculator() [1/2]

```
UltraCold::MKLWrappers::DFtCalculator::DFtCalculator (
            Vector< std::complex< double > > & forward_domain_vector,
            Vector< std::complex< double > > & backward_domain_vector )
```

Constructor for complex-complex transforms.

Parameters

forward_domain_vector	*Vector<std::complex<double>>* Vector defined on the forward domain, i.e. whose domain lives in real space
backward_domain_vector	*Vector<std::complex<double>>* Vector defined on the backward domain, i.e. whose domain lives in Fourier space

The constructor initializes the appropriate descriptor type.

8.3.2.2 DFtCalculator() [2/2]

```
UltraCold::MKLWrappers::DFtCalculator::DFtCalculator (
            Vector< double > & forward_domain_vector,
            Vector< std::complex< double > > & backward_domain_vector )
```

Constructor for real-complex transforms.

Parameters

forward_domain_vector	*Vector<double>* Vector defined on the forward domain, i.e. whose domain lives in real space
backward_domain_vector	*Vector<std::complex<double>>* Vector defined on the backward domain, i.e. whose domain lives in Fourier space

The constructor initializes the appropriate descriptor type.

Note

Since the Fourier transform of a real function is conjugate symmetric, only half of the values needs to be stored. Using Intel's MKL DFT functions, the halved dimension is the last one.

8.3.3 Member Function Documentation

8.3.3.1 compute_backward()

```
void UltraCold::MKLWrappers::DFtCalculator::compute_backward ( )
```

Calculate a backward transform.

The transform is alreay normalized.

8.4 UltraCold::GPSolvers::DipolarGPSolver Class Reference

Class to solve the Gross-Pitaevskii equation for a dipolar Bose gas with the Lee-Huang-Yang correction.

```
#include <gp_solvers.hpp>
```

Public Member Functions

- DipolarGPSolver (Vector< double > &x, Vector< double > &y, Vector< double > &z, Vector< std::complex< double > > &psi0, Vector< double > &Vext, double scattering_length, double dipolar_length)

  Constructor for a DipolarGPSolver in three space dimensions.
- void reinit (Vector< double > &Vext, Vector< std::complex< double > > &psi)

  Reinitialize the solver with new external potential and initial condition.
- std::tuple< Vector< std::complex< double > >, double > run_gradient_descent (int max_num_iter, double tolerance, double alpha, double beta, std::ostream &output_stream)

  Calculates a ground-state solution to the stationary, extended Gross-Pitaevskii equation with Lee-Huang-Yang correction.
- virtual void run_operator_splitting (int number_of_time_steps, double time_step, std::ostream &output_↩ stream)

  Solve the extended Gross-Pitaevskii equation using simple operator splitting.
- virtual void **run_operator_splitting** (int, double, double, std::ostream &)

  Useful possible overload.
- virtual void **run_operator_splitting** (int, double, double, double, std::ostream &)

  Useful possible overload.
- virtual void **run_operator_splitting** (int, double, double, double, double, std::ostream &)

  Useful possible overload.

Protected Member Functions

- virtual void write_gradient_descent_output (size_t iteration_number, std::ostream &output_stream)
  Write output at each step of the gradient descent iterations.
- virtual void write_operator_splitting_output (size_t iteration_number, std::ostream &output_stream)
  Write some output at each time step.
- virtual void **write_operator_splitting_output** (size_t, double, std::ostream &)
  Useful possible overload.
- virtual void **write_operator_splitting_output** (size_t, double, double, std::ostream &)
  Useful possible overload.
- virtual void **write_operator_splitting_output** (size_t, double, double, double, std::ostream &)
  Useful possible overload.
- virtual void **solve_step_1_operator_splitting** (MKLWrappers::DFtCalculator &)
  Solve step-1 of operator splitting.
- virtual void **solve_step_1_operator_splitting** (MKLWrappers::DFtCalculator &, double)
  Useful possible overload.
- virtual void **solve_step_1_operator_splitting** (MKLWrappers::DFtCalculator &, double, double)
  Useful possible overload.
- virtual void **solve_step_1_operator_splitting** (MKLWrappers::DFtCalculator &, double, double, double)
  Useful possible overload.
- void **solve_step_2_operator_splitting** (MKLWrappers::DFtCalculator &)
  Solve step-2 of operator splitting.

Protected Attributes

- Vector< std::complex< double > > **psi**
- Vector< double > **Vext**
- Vector< double > **x**
- Vector< double > **y**
- Vector< double > **z**
- Vector< double > **kx**
- Vector< double > **ky**
- Vector< double > **kz**
- Vector< double > **kmod2**
- Vector< std::complex< double > > **psitilde**
- Vector< std::complex< double > > **hpsi**
- Vector< double > **Vtilde**
- Vector< double > **Phi_dd**
- Vector< std::complex< double > > **Phi_tilde**
- int **nx**
- int **ny**
- int **nz**
- double **dx** = 1.0
- double **dy** = 1.0
- double **dz** = 1.0
- double **dv** = 1.0
- double **chemical_potential**
- double **scattering_length**
- double **dipolar_length**
- double **epsilon_dd**
- double **gamma_epsilon_dd**
- double **residual**
- double **initial_norm**
- double **norm**
- double **time_step**
- std::complex< double > **ci** ={0.0,1.0}
- int **last_iteration_number**

8.4.1 Detailed Description

Class to solve the Gross-Pitaevskii equation for a dipolar Bose gas with the Lee-Huang-Yang correction.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

This class allows to solve the extended Gross-Pitaevskii equation for a dipolar Bose gas

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \mathcal{H}(\mathbf{r})\,\Psi(\mathbf{r},t)\,,$$

where the Hamiltonian H is

$$\mathcal{H}(\mathbf{r}) = -\frac{\hbar^2}{2m}\nabla^2 + V_{\text{ext}}(\mathbf{r}) + g|\Psi(\mathbf{r},t)|^2 + \gamma(\varepsilon_{dd})|\Psi(\mathbf{r},t)|^3 + \int d\mathbf{r'}V_{dd}(\mathbf{r}-\mathbf{r'})|\Psi(\mathbf{r'},t)|^2\,,$$

with $g = 4\pi\hbar^2 a/m$ the coupling constant fixed by the s-wave scattering length a and $V_{dd}(\mathbf{r}_i - \mathbf{r}_j) = \frac{\mu_0\mu^2}{4\pi}\frac{1-3\cos^2\theta}{|\mathbf{r}_i-\mathbf{r}_j|^3}$ the dipole-dipole potential, being μ_0 the magnetic permeability in vacuum, μ the magnetic dipole moment and θ the angle between the vector distance between dipoles and the polarization direction, which we choose as the x-axis. In the absence of trapping, the system can be fully characterised by the single parameter $\varepsilon_{dd} = \mu_0\mu^2/(3g) = a_{dd}/a$, i.e., the ratio between the strength of the dipolar and the contact interaction, eventually written in terms of the dipolar length a_{dd} and the scattering length a. The third term of the Hamiltonian corresponds to the local density approximation of the beyond-mean-field Lee-Huang-Yang (LHY) correction with

$$\gamma(\varepsilon_{dd}) = \frac{16}{3\sqrt{\pi}}ga^{\frac{3}{2}}\,\text{Re}\left[\int_0^\pi d\theta\sin\theta[1+\varepsilon_{dd}(3\cos^2\theta-1)]^{\frac{5}{2}}\right].$$

Experimental measurements and microscopic Monte Carlo calculations have confirmed that the LHY term is an accurate correction to the mean-field theory given by the Gross-Pitaevskii equation in dipolar gases.
This class can be used to solve the extended Gross-Pitaevskii equation both for ground-state configurations as well as for the dynamics, on a cartesian mesh with periodic boundary conditions. The basic usage of the class is as follows:

```
DipolarGPSolver dipolar_gp_solver(x,
                                  y,
                                  z,
                                  psi0,
                                  Vext,
                                  scattering_length,
                                  dipolar_length);
// for ground state calculations
std::tie(psi,chemical_potential) = dipolar_gp_solver.run_gradient_descent(maximum_number_of_iterations,
                                                                          tolerance,
                                                                          alpha,
                                                                          beta,
                                                                          std::cout);
// or
dipolar_gp_solver.run_operator_splitting(number_of_time_steps,time_step); // for the dynamics
```

In the constructor, you need to provide the three cartesian axis defining the mesh on which the equation is going to be solved, the initial wave function (all the methods are iterative), the external potential, the s-wave scattering length a and the dipolar length a_{dd}, both in appropriate units.
The algorithms used and their working principles are the same as those used for a non-dipolar Bose gas, described in the documentation of the GPSolver class. See that documentation for details.

Note

Several member functions of this class take advantage of OpenMP parallelization. For optimal performance, be sure to run

```
$ export OMP_NUM_THREADS=<number of physical cores on the machine used>
```

on the shell before the program execution.

Warning

The extended Gross-Pitaevskii equation is solved in its a-dimensional form, which, in the case of, for example, harmonic trapping, is given by

$$i\frac{\partial\psi(\mathbf{r},t)}{\partial t} = \left[-\frac{\nabla^2}{2} + \left(\frac{1}{2}\left(\frac{x}{a_x^2}\right)^2 + \frac{1}{2}\left(\frac{y}{a_y^2}\right)^2 + \frac{1}{2}\left(\frac{z}{a_z^2}\right)^2\right) + 4\pi a|\psi(\mathbf{r},t)|^2 + 3a_{dd}\int d\mathbf{r'}\frac{1-3\cos^2\theta}{|\mathbf{r}-\mathbf{r'}|^3}|\psi(\mathbf{r'},t)|^2 + \frac{64\sqrt{\pi}}{3}a^{5/2}$$

where $a_{x,y,z} = \sqrt{\frac{\hbar}{m\omega_{x,y,z}}}$ and all the lengths (including the s-wave and the dipolar length) are in units of the harmonic oscillator length $a_{ho} = (a_x a_y a_z)^{1/3}$

8.4.2 Constructor & Destructor Documentation

8.4.2.1 DipolarGPSolver()

```
UltraCold::GPSolvers::DipolarGPSolver::DipolarGPSolver (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & z,
            Vector< std::complex< double > > & psi_0,
            Vector< double > & Vext,
            double scattering_length,
            double dipolar_length )
```

Constructor for a DipolarGPSolver in three space dimensions.

Parameters

x	*Vector<double>* representing the x-axis of the Cartesian frame on which the Gross-Pitaevskii equation in two space dimensions will be solved
y	*Vector<double>* representing the y-axis of the Cartesian frame on which the Gross-Pitaevskii equation in two space dimensions will be solved
z	*Vector<double>* representing the z-axis of the Cartesian frame on which the Gross-Pitaevskii equation in two space dimensions will be solved
psi_0	*Vector<std::complex<double>>* representing the initial wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units
epsilon_dd	*double* ratio between scattering and dipolar length

8.4.3 Member Function Documentation

8.4.3.1 reinit()

```
void UltraCold::GPSolvers::DipolarGPSolver::reinit (
            Vector< double > & Vext,
            Vector< std::complex< double > > & psi_0 )
```

Reinitialize the solver with new external potential and initial condition.

Parameters

Vext	*Vector<double>* the new external potential.
psi_0	*Vector<std::complex<double>>* the new initial wave function.

Warning

This function does not perform any bound check, hence you must be careful to pass Vectors with the same dimensionality and extents as those passed to the constructor.

8.4.3.2 run_gradient_descent()

```
std::tuple< Vector< std::complex< double > >, double > UltraCold::GPSolvers::DipolarGPSolver
::run_gradient_descent (
            int max_num_iter,
            double tolerance,
            double alpha,
            double beta,
            std::ostream & output_stream )
```

Calculates a ground-state solution to the stationary, extended Gross-Pitaevskii equation with Lee-Huang-Yang correction.

Parameters

max_num_iter	*int* the maximum number of gradient descent iterations
tolerance	*double* the maximum norm of the residual, below which the algorithm is considered as converged
alpha	*double* the step-length of the gradient-descent iterations.
beta	*double* acceleration step for the heavy-ball method.
output_stream	*std::ostream* stream to which eventual text output can be passed

Returns

std::tuple<Vector<std::complex<double>>,double> representing the calculated ground-state wave function and chemical potential. Can be recovered by using std::tie(psi,chemical_potential).

This member function allows to solve the extended Gross-Pitaevskii equation describing a dipolar Bose gas, with the addition of the Lee-Huang-Yang correction, using the same gradient-descend algorithm, accelerated via the heavy-ball method, used in the corresponding member function of the class GPSolver. See that documentation for more details.

8.4.3.3 run_operator_splitting()

```
void UltraCold::GPSolvers::DipolarGPSolver::run_operator_splitting (
            int number_of_time_steps,
            double time_step,
            std::ostream & output_stream )  [virtual]
```

Solve the extended Gross-Pitaevskii equation using simple operator splitting.

Parameters

number_of_time_steps	*int* The total number of time-steps to be performed
time_step	*double* time step in appropriate units
output_stream	*std::ostream* stream to which eventual text output can be passed

This member function solves the (a-dimensional) time-dependent extended Gross-Pitaevskii equation

$$i\frac{\partial\psi}{\partial t} = \left[\frac{-\nabla^2}{2} + V_{ext}(\mathbf{r}) + g|\psi|^2\right]\psi$$

for the description of a dipolar Bose gas using the classic operator splitting technique. The idea is the same as the corresponding member class of GPSolver. See that documentation for details.

8.4.3.4 write_gradient_descent_output()

```
void UltraCold::GPSolvers::DipolarGPSolver::write_gradient_descent_output (
            size_t iteration_number,
            std::ostream & output_stream )  [protected], [virtual]
```

Write output at each step of the gradient descent iterations.

This function can (and should) be overridden in derived classes.

Parameters

iteration_number	*size_t* current iteration number
output_stream	*std::ostream* stream to which eventual text output can be passed

Warning

Writing output data files at each gradient descent step may be useful, but it is also very expensive. Override this member function with care!

8.4.3.5 write_operator_splitting_output()

```
void UltraCold::GPSolvers::DipolarGPSolver::write_operator_splitting_output (
            size_t iteration_number,
            std::ostream & output_stream )  [protected], [virtual]
```

Write some output at each time step.

Parameters

iteration_number	*size_t* current iteration number
output_stream	*std::ostream* stream to which eventual text output can be passed

8.5 UltraCold::GPSolvers::GPSolver Class Reference

Class to solve the Gross-Pitaevskii equation.

`#include <gp_solvers.hpp>`

Inherited by myGPSolver.

Public Member Functions

- GPSolver (Vector< double > &x, Vector< std::complex< double > > &psi_0, Vector< double > &Vext, double scattering_length)

  Constructor for a GPSolver in one space dimension.
- GPSolver (Vector< double > &x, Vector< double > &y, Vector< std::complex< double > > &psi_0, Vector< double > &Vext, double scattering_length)

  Constructor for a GPSolver in two space dimensions.
- GPSolver (Vector< double > &x, Vector< double > &y, Vector< double > &z, Vector< std::complex< double > > &psi_0, Vector< double > &Vext, double scattering_length)

  Constructor for a GPSolver in three space dimensions.
- void reinit (Vector< double > &Vext, Vector< std::complex< double > > &psi_0)

  Reinitialize the solver with new external potential and initial condition.
- std::tuple< Vector< std::complex< double > >, double > run_gradient_descent (int max_num_iter, double tolerance, double alpha, double beta, std::ostream &output_stream)

  Calculates a ground-state solution to the stationary Gross-Pitaevskii equation.
- virtual void run_operator_splitting (int number_of_time_steps, double time_step, std::ostream &output_stream)

  Solve the Gross-Pitaevskii equation using simple operator splitting.
- virtual void **run_operator_splitting** (int, double, double, std::ostream &)

  Useful possible overload.
- virtual void **run_operator_splitting** (int, double, double, double, std::ostream &)

  Useful possible overload.
- virtual void run_operator_splitting (int, double, double, double, double, std::ostream &)

  Useful possible overload.

Protected Member Functions

- virtual void write_gradient_descent_output (size_t iteration_number, std::ostream &output_stream)

  Write output at each step of the gradient descent iterations.
- virtual void write_operator_splitting_output (size_t iteration_number, std::ostream &output_stream)

  Write some output at each time step.
- virtual void write_operator_splitting_output (size_t, double, std::ostream &)

  Useful possible overload.
- virtual void **write_operator_splitting_output** (size_t, double, double, std::ostream &)

  Useful possible overload.
- virtual void **write_operator_splitting_output** (size_t, double, double, double, std::ostream &)

  Useful possible overload.
- virtual void **solve_step_1_operator_splitting** ()

  Solve step-1 of operator splitting.
- virtual void solve_step_1_operator_splitting (double)

  Useful possible overload.
- virtual void **solve_step_1_operator_splitting** (double, double)

  Useful possible overload.
- virtual void **solve_step_1_operator_splitting** (double, double, double)

  Useful possible overload.
- void **solve_step_2_operator_splitting** (MKLWrappers::DFtCalculator &)

  Solve step-2 of operator splitting.

Protected Attributes

- Vector< std::complex< double > > **psi**
- Vector< double > **Vext**
- Vector< double > **x**
- Vector< double > **y**
- Vector< double > **z**
- Vector< double > **kx**
- Vector< double > **ky**
- Vector< double > **kz**
- Vector< double > **kmod2**
- Vector< std::complex< double > > **psitilde**
- Vector< std::complex< double > > **hpsi**
- int **nx**
- int **ny**
- int **nz**
- double **dx** = 1.0
- double **dy** = 1.0
- double **dz** = 1.0
- double **dv** = 1.0
- double **chemical_potential**
- double **scattering_length**
- double **residual**
- double **initial_norm**
- double **norm**
- double **time_step**
- std::complex< double > **ci** ={0.0,1.0}
- int **last_iteration_number**

8.5.1 Detailed Description

Class to solve the Gross-Pitaevskii equation.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

This class allows to solve the Gross-Pitaevskii equation

$$i\hbar\frac{\partial\psi}{\partial t} = \left[\frac{-\hbar^2\nabla^2}{2m} + V_{ext}(\mathbf{r}) + g|\psi|^2\right]\psi$$

both for ground-state configurations as well as for the dynamics, on a cartesian mesh with periodic boundary conditions. The basic usage of the class is as follows:

```
GPSolver gp_solver(x,y,z,psi0,Vext,scattering_length);
// for ground state calculations
std::tie(psi,chemical_potential) = gp_solver.run_gradient_descent(maximum_number_of_iterations,
                                                                  tolerance,
                                                                  alpha,
                                                                  beta,
                                                                  std::cout);
// or
gp_solver.run_operator_splitting(number_of_time_steps,time_step); // for the dynamics
```

In the constructor, you need to provide the (cartesian) axis along which you want to solve the equation (of course, 1, 2 or 3), the initial wave function (all the methods are iterative) and the external potential. All of these must be properly initialized before passing them to the solver class.
For ground-state calculations, it is possible to generate some output at each step of the gradient-descent iterations. By default, the GPSolver::run_gradient_descent() just writes, in the standard output, the current iteration number, chemical potential and norm of the residual. This default behavior can however be customized by overriding the GPSolver::write_gradient_descent_output() member function.
A similar behavior is the default also for dynamics calculations, i.e. for the member function GPSolver::run_operator_splitting(). In this case, the default is just to write the current time step and current time. For examples of usage and customization via overloading, see the example 1 in the `examples` folder.

Note

Several member functions of this class take advantage of OpenMP parallelization. For optimal performance, be sure to run

```
$ export OMP_NUM_THREADS=<number of physical cores on the machine used>
```

on the shell before the program execution.

Warning

The Gross-Pitaevskii equation is solved in its a-dimensional form. In the case of a harmonic potential in one space dimension it has the form

$$i\frac{\partial\psi}{\partial t} = \left[-\frac{1}{2}\frac{\partial^2}{\partial x^2} + \frac{1}{2}x^2 + 4\pi a|\psi|^2\right]\psi$$

where a is the scattering length, which here must be given in harmonic units. Be sure to provide the axis, initial wave function, and external potential initialized properly. See the examples in the examples folder.

8.5.2 Constructor & Destructor Documentation

8.5.2.1 GPSolver() [1/3]

```
UltraCold::GPSolvers::GPSolver::GPSolver (
            Vector< double > & x,
            Vector< std::complex< double > > & psi_0,
            Vector< double > & Vext,
            double scattering_length )
```

Constructor for a GPSolver in one space dimension.

Parameters

x	*Vector<double>* representing the cartesian axis on which the Gross-Pitaevskii equation in one space dimension will be solved
psi_0	*Vector<std::complex<double>>* representing the initial wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units

8.5.2.2 GPSolver() [2/3]

```
UltraCold::GPSolvers::GPSolver::GPSolver (
            Vector< double > & x,
            Vector< double > & y,
            Vector< std::complex< double > > & psi_0,
            Vector< double > & Vext,
            double scattering_length )
```

Constructor for a GPSolver in two space dimensions.

Parameters

x	*Vector<double>* representing the x-axis of the Cartesian frame on which the Gross-Pitaevskii equation in two space dimensions will be solved
y	*Vector<double>* representing the y-axis of the Cartesian frame on which the Gross-Pitaevskii equation in two space dimensions will be solved
psi_0	*Vector<std::complex<double>>* representing the initial wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units

8.5.2.3 GPSolver() [3/3]

```
UltraCold::GPSolvers::GPSolver::GPSolver (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & z,
```

```
            Vector< std::complex< double > > & psi_0,
            Vector< double > & Vext,
            double scattering_length )
```

Constructor for a GPSolver in three space dimensions.

Parameters

x	*Vector<double>* representing the x-axis of the Cartesian frame on which the Gross-Pitaevskii equation in two space dimensions will be solved
y	*Vector<double>* representing the y-axis of the Cartesian frame on which the Gross-Pitaevskii equation in two space dimensions will be solved
z	*Vector<double>* representing the z-axis of the Cartesian frame on which the Gross-Pitaevskii equation in two space dimensions will be solved
psi_0	*Vector<std::complex<double>>* representing the initial wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units

8.5.3 Member Function Documentation

8.5.3.1 reinit()

```
void UltraCold::GPSolvers::GPSolver::reinit (
            Vector< double > & Vext,
            Vector< std::complex< double > > & psi_0 )
```

Reinitialize the solver with new external potential and initial condition.

Parameters

Vext	*Vector<double>* the new external potential.
psi_0	*Vector<std::complex<double>>* the new initial wave function.

Warning

This function does not perform any bound check, hence you must be careful to pass Vectors with the same dimensionality and extents as those passed to the constructor.

8.5.3.2 run_gradient_descent()

```
std::tuple< Vector< std::complex< double > >, double > UltraCold::GPSolvers::GPSolver::run_gradient_descent (
            int max_num_iter,
```

```
double tolerance,
double alpha,
double beta,
std::ostream & output_stream )
```

Calculates a ground-state solution to the stationary Gross-Pitaevskii equation.

Parameters

max_num_iter	*int* the maximum number of gradient descent iterations
tolerance	*double* the maximum norm of the residual, below which the algorithm is considered as converged
alpha	*double* the step-length of the gradient-descent iterations.
beta	*double* acceleration step for the heavy-ball method.
output_stream	*std::ostream* stream to which eventual text output can be passed

Returns

std::tuple<Vector<std::complex<double>>,double> representing the calculated ground-state wave function and chemical potential. Can be recovered by using std::tie(psi,chemical_potential).

The Gross-Pitaevskii equation allows to obtain information on the ground-state properties of an ultra-cold bosonic system by searching for stationary solutions of the form $\psi(\mathbf{r},t) = \psi_0(\mathbf{r})e^{-i\mu t/\hbar}$, obtaining the stationary Gross-Pitaevskii equation

$$\mu\psi_0 = \left[\frac{-\hbar^2\nabla^2}{2m} + V_{ext}(\mathbf{r}) + g|\psi_0|^2\right]\psi_0$$

where g is related to the s-wave scattering length a by $g = \frac{4\pi\hbar^2 a}{2m}$. Solving this for the smallest eigenvalue μ, which represents the chemical potential, gives access to the ground-state configuration of the system.
In order to solve this equation, one can notice that it can be obtained from a constrained minimization formulation of the problem, in particular by requiring that the ground-state order parameter of the system is the exact minimizer of the mean-field energy functional

$$E[\psi] = \int d\mathbf{r}\left[\psi^*(\mathbf{r})\left(\frac{-\hbar^2\nabla^2}{2m} + V_{ext}(\mathbf{r})\right)\psi(\mathbf{r})\right] + \frac{g}{2}\int d\mathbf{r}|\psi(\mathbf{r})|^4$$

under the constraint of a fixed number of particles $\int d\mathbf{r}|\psi(\mathbf{r})|^2 = N$. This allows also to introduce the chemical potential μ as the Lagrange multiplier fixing the total number of particles. One can hence find the ground state order parameter and chemical potential using, for example, a `gradient descent` method, which is the one implemented in this function.
The idea is to start from a guess solution ψ_0 and generate a sequence of iterates $\{\psi_n\}_{\{n=0,\dots,\infty\}}$ that terminates when one is sufficiently confident to have reached a (hopefully global) minimizer of the mean-field energy functional with good accuracy. In particular, a good stopping criterion consists in fixing a tolerance threshold ϵ (which, for this function, is a user-defined input parameter) for the norm of the residual, i.e. $||\hat{H}\psi_n\mu_n\psi_n||^2 \leq \epsilon$, where $\hat{H}$ is the mean-field Hamiltonian of the system and the estimate μ_n of the chemical potential μ at iteration n can be calculated as $\mu_n = \langle\psi_n|\hat{H}|\psi_n\rangle/\langle\psi_n|\psi_n\rangle$.
In deciding how to move from one iterate ψ_n to the next ψ_{n+1}, line search algorithms like the gradient descent method use information about the functional $E[\psi]$ at ψ_n, and possibly also from earlier iterates $\psi_0, \psi_1, \dots, \psi_{n-1}$. The update criterion should be that the energy functional is smaller in ψ_{n+1} then in ψ_n. One thus generates a sequence $\psi_{n+1} = \psi_n + \alpha\chi_n$ such that $E[\psi_{n+1}] < E[\psi_n] < \cdots < E[\psi_0]$ until the stopping criterion is satisfied. The update "direction" χ_n must thus be chosen to be a descent direction, i.e. a direction along which the functional $E[\psi]$ decreases. The step-length α should instead be (ideally) chosen in such a way that the decrease in the energy functional is, at each iteration step, the maximum possible. Since this is not, in general, an easy task, in this function we accept the compromise to choose the step length α empirically as an input parameter at the beginning of the iteration procedure. This also implies that the user may need to run the function a few times before finding an optimal value for α.
Coming back to the choice of χ_n, the gradient descent method consists in choosing such descent direction as the

opposite of the gradient of the functional $E[\psi]$ calculated in ψ_n . So, in this case, the descent direction is (minus) the functional derivative of the energy functional with respect to ψ^* evaluated at ψ_n , i.e.

$$\chi_n = -\frac{\delta E[\psi_n]}{\delta \psi^*} = -\left[\frac{-\hbar^2\nabla^2}{2m} + V_{ext}(\mathbf{r}) + g|\psi_n|^2\right]\psi_n$$

The ground state wave function obtained from this algorithm is normalized in such a way to preserve the initial norm, i.e. the norm of the initial wave function provided. Such normalization condition, fixing the L^2 norm of the ground-state wave-function ψ should be included in the iteration procedure by introducing a Lagrange multiplier and minimizing the corresponding lagrangian functional. In practice, it is much cheaper to normalize "by hand" each ψ_n obtained via the gradient descent iteration, by fixing

$$\psi_{n+1}^{(1)} = \psi_n + \alpha\chi_n$$

$$\psi_{n+1} = \sqrt{\frac{N}{\int d\mathbf{r}|\psi_{n+1}^{(1)}|^2}}\psi_{n+1}^{(1)}$$

where N is the initial norm. Notice that, physically, this represents the total number of particles in the system, and as such must be an integer. Hence, the initial wave function must be properly normalized before the gradient-descent iterations start.
Finally, this function employs an acceleration algorithm, known as the heavy ball method , in order to speed up the convergence of the sequence ψ_n. The method consists in adding a "momentum" term into the gradient-descent iterations, in order to make larger steps if the descent direction does not change very much, and smaller steps if it changes a lot. In practice, what one does is just to modify the gradient-descent expression in $\psi_{n+1} = \psi_n + \alpha\chi_n + \beta(\psi_n - \psi_{n-1})$ where, again, β is a parameter chosen empirically and given to this function as input.

8.5.3.3 run_operator_splitting() [1/2]

```
void UltraCold::GPSolvers::GPSolver::run_operator_splitting (
            int number_of_time_steps,
            double time_step,
            std::ostream & output_stream )  [virtual]
```

Solve the Gross-Pitaevskii equation using simple operator splitting.

Parameters

number_of_time_steps	*int* The total number of time-steps to be performed
time_step	*double* time step in appropriate units
output_stream	*std::ostream* stream to which eventual text output can be passed

This member function solves the (a-dimensional) time-dependent Gross-Pitaevskii equation

$$i\frac{\partial\psi}{\partial t} = \left[\frac{-\nabla^2}{2} + V_{ext}(\mathbf{r}) + g|\psi|^2\right]\psi$$

using the classic operator splitting technique.
The general idea of operator-splitting methods is to consider an initial value problem

$$y' = \mathrm{A}y + \mathrm{B}y$$

where A and B are differential operators, and solve the equation considering the actions of the two operators separately. There is a vast literature on operator-splitting approaches for the solution of differential equations, and different methods with a higher or lower level of accuracy. In the case of the Gross-Pitaevskii equation, things are even more simplified by the fact that part of the method implies steps that can be solved *exactly*.
After choosing a time-step Δt, the operator splting scheme consists in the following steps

For $n = 0, 1, \ldots,$ number$_o f_t ime_s teps$number$_o f_t ime_s teps_{\text{number}_o f_t ime_s teps}$Step 1: solve $y'_1 = \text{A}y_1$ in $[t_n, t_n+\Delta$

The steps imply the solution of an ordinary differential equation. In the case of the Gross-Pitaevskii equation, a good choice of the operators A and B is the following

$$A = V_{ext} + g|\psi(t)|^2$$
$$B = \frac{-\nabla^2}{2}$$

In this case, step 1, although involve the solution of a non-linear differential equation, can be solved **analytically**. In fact, it is easy to show that the equation

$$i\frac{d}{dt}\psi(t) = V_{ext} + g|\psi(t)|^2$$

preserves the norm ψ in time, and hence an explicit solution of this equation is

$$\psi(t + \Delta t) = e^{-i\Delta t\left(V_{ext}+g|\psi(t)|^2\right)}\psi(t)$$

Moreover, also the second step of the method can be solved analytically, provided that one imposes *periodic boundary conditions*. In fact, given the equation

$$i\frac{d}{dt}\psi(t) = \frac{-\nabla^2}{2}\psi(t)$$

and taking the Fourier transform (in space) at both sides, one finds

$$i\frac{d}{dt}\tilde{\psi}(t) = \frac{k^2}{2}\tilde{\psi}(t)$$

which can again be solved exactly as

$$\tilde{\psi}(t + \Delta t) = e^{-i\Delta t\frac{k^2}{2}}\tilde{\psi}(t)$$

Finally, an inverse Fourier transform allows to recover the solution back in real space, ready to take another time step.
To summarize, this member function solves the Gross-Pitaevskii equation using classic operator splitting via the following steps

- Step 1) Set $\psi(t_n + \Delta t) = e^{-i\Delta t\left(V_{ext}+g|\psi(t_n)|^2\right)}\psi(t_n)$;
- Step 2)
  - 2.1) Take the Fourier transform of $\psi(t_n + \Delta t)$, and call it $\tilde{\psi}(t_n)$;
  - 2.2) Set $\tilde{\psi}(t_n + \Delta t) = e^{-i\Delta t\frac{k^2}{2}}\tilde{\psi}(t_n)$
  - 2.3) Take the inverse Fourier transform of $\tilde{\psi}(t_n + \Delta t)$, and obtain $\psi(t_n + \Delta t)$

Notice that all these steps can be computed *locally*, meaning that we don't need additional vectors to store intermediate values of ψ.

8.5.3.4 run_operator_splitting() [2/2]

```
void UltraCold::GPSolvers::GPSolver::run_operator_splitting (
            int ,
            double ,
            double ,
            double ,
            double ,
            std::ostream &  )  [virtual]
```

Useful possible overload.

Reimplemented in myGPSolver.

8.5.3.5 solve_step_1_operator_splitting()

```
void UltraCold::GPSolvers::GPSolver::solve_step_1_operator_splitting (
            double  )  [protected], [virtual]
```

Useful possible overload.

Reimplemented in myGPSolver.

8.5.3.6 write_gradient_descent_output()

```
void UltraCold::GPSolvers::GPSolver::write_gradient_descent_output (
            size_t iteration_number,
            std::ostream & output_stream )  [protected], [virtual]
```

Write output at each step of the gradient descent iterations.

This function can (and should) be overridden in derived classes.

Parameters

iteration_number	*size_t* current iteration number
output_stream	*std::ostream* stream to which eventual text output can be passed

Warning

Writing output data files at each gradient descent step may be useful, but it is also very expensive. Override this member function with care!

8.5.3.7 write_operator_splitting_output() [1/2]

```
void UltraCold::GPSolvers::GPSolver::write_operator_splitting_output (
            size_t iteration_number,
            std::ostream & output_stream )  [protected], [virtual]
```

Write some output at each time step.

Parameters

iteration_number	*size_t* current iteration number
output_stream	*std::ostream* stream to which eventual text output can be passed

8.5.3.8 write_operator_splitting_output() [2/2]

```
void UltraCold::GPSolvers::GPSolver::write_operator_splitting_output (
            size_t ,
            double ,
            std::ostream &  )  [protected], [virtual]
```

Useful possible overload.

Reimplemented in myGPSolver.

8.6 UltraCold::Tools::HardwareInspector Class Reference

Class to detect hardware capabilities.

```
#include <hardware_inspector.hpp>
```

Public Member Functions

- HardwareInspector ()

  Constructor of class HardwareInspector.
- int get_number_of_processors ()

  Returns the total number of processors present in the machines.
- int get_number_of_available_processors ()

  Returns the number of available processors.
- void **print_cpu_information** ()

  Print all available hardware information on the screen.

8.6.1 Detailed Description

Class to detect hardware capabilities.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

The HardwareInspector class provides methods to check hardware information on the current machine, such as the number of available cores, or the total amount of available memory. It also helps in tuning the performance of UltraCold.

8.6.2 Constructor & Destructor Documentation

8.6.2.1 HardwareInspector()

```
UltraCold::Tools::HardwareInspector::HardwareInspector ( )
```

Constructor of class HardwareInspector.

The constructor actually retrieves the hardware information on the machines and initializes the corresponding member variables.

8.6.3 Member Function Documentation

8.6.3.1 get_number_of_available_processors()

```
int UltraCold::Tools::HardwareInspector::get_number_of_available_processors ( )
```

Returns the number of available processors.

Notice that this number may be different from the total number of processors present on the machine.

Returns

the number of available processors

8.6.3.2 get_number_of_processors()

```
int UltraCold::Tools::HardwareInspector::get_number_of_processors ( )
```

Returns the total number of processors present in the machines.

Notice that this number may be different from the actual number of processors available for the program.

Returns

the total number of processors present on the machine

8.7 UltraCold::Tools::InputParser Class Reference

Class to read input parameters from files.

```
#include <input_parser.hpp>
```

Public Member Functions

- InputParser (char *input_file_name)
  Constructor of class InputParser.
- ~InputParser ()
  Destructor.
- void read_input_file ()
  Member function to parse the input file.
- int retrieve_int (const char *variable_name)
  Cast an element retrieved to int.
- double retrieve_double (const char *variable_name)
  Cast an element retrieved to double.
- bool retrieve_bool (const char *variable_name)
  Cast an element retrieved to bool.

8.7.1 Detailed Description

Class to read input parameters from files.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

The InputParser class provides an interface to input parameter files. The class can be used as follows. Suppose you have a file input_file.prm containing the following text

```
# Define the mesh parameters

xmax = 10.0 # Size of the mesh along the x-axis, in micrometers.
ymax = 10.0 # Size of the mesh along the y-axis, in micrometers.
zmax = 10.0 # Size of the mesh along the z-axis, in micrometers.

nx = 100 # number of points along the x-axis
ny = 100 # number of points along the y-axis
nz = 100 # number of points along the z-axis

# Define interaction parameters

scattering length = 100.0 # Scattering length in micrometers

# Define the run mode

calculate ground state = true # If true, calculate a ground state solution of the eGPE
```

Comments are defined by an hashtag (#) and ignored by the class.
Every line containing an equal (=) sign is interpreted as a line defining an input. Every character on the left of the equal sign is interpreted as the name of the variable (without blanks), while what follows the equal sign on the right is interpreted as the value of the defined variable.
In order to retrieve the value of these variables and use them in the code, it is necessary to use the appropriate member function `<variable type> InputParser::retrieve_<type>("variable name")` , which will cast the retrieved variable (which is first interpreted as an `std::string`) to the desired `<type>` .
So, an example code block that reads the input file defined above and initializes the variables *xmax*, *ymax*, *zmax*, *scattering length*, and *run in imaginary time*, is the following

```
InputParser ip("input_file.prm");
ip.read_input_file();
const double xmax = ip.retrieve_double("xmax");
const double ymax = ip.retrieve_double("ymax");
const double zmax = ip.retrieve_double("zmax");
const int nx = ip.retrieve_int("nx");
const int ny = ip.retrieve_int("ny");
const int nz = ip.retrieve_int("nz");
const double scattering_length = ip.retrieve_double("scattering length"));
const bool calc_ground_state = ip.retrieve_bool("calculate ground state");
```

8.7.2 Constructor & Destructor Documentation

8.7.2.1 InputParser()

```
UltraCold::Tools::InputParser::InputParser (
            char * input_file_name )
```

Constructor of class InputParser.

Parameters

input_file_name	*char* the name of the input file

The constructor tries to open the file `input_file_name` . If the file is not found, it will print an error message an terminate the execution of the program. If instead the file is found, the constructor initializes an `std::ifstream` through which the file can be read.

8.7.2.2 ~InputParser()

```
UltraCold::Tools::InputParser::~InputParser ( )
```

Destructor.

The destructor simply closes the `std::ifstream` used to read the input file.

8.7.3 Member Function Documentation

8.7.3.1 read_input_file()

```
void UltraCold::Tools::InputParser::read_input_file ( )
```

Member function to parse the input file.

This function parses the input file, ignoring all comments (i.e., everything coming after an hashtag (#)), and filling an `std::map<std::string,std::string>` with the pairs name=value defined by lines containing an equal (=) sign.

8.7.3.2 retrieve_bool()

```
bool UltraCold::Tools::InputParser::retrieve_bool (
            const char * requested_element )
```

Cast an element retrieved to bool.

Parameters

requested_element	∗char∗∗ Name of the element to be retrieved.

Returns

the requested element, cast to bool

8.7.3.3 retrieve_double()

```
double UltraCold::Tools::InputParser::retrieve_double (
            const char * requested_element )
```

Cast an element retrieved to double.

Parameters

requested_element	∗char∗∗ Name of the element to be retrieved.

Returns

the requested element, cast to double

8.7.3.4 retrieve_int()

```
int UltraCold::Tools::InputParser::retrieve_int (
            const char * requested_element )
```

Cast an element retrieved to int.

Parameters

requested_element	∗char∗∗ Name of the element to be retrieved.

Returns

the requested element, cast to int

8.8 myGPSolver Class Reference

Inherits UltraCold::GPSolvers::GPSolver.

Public Member Functions

- void run_operator_splitting (int number_of_time_steps, double time_step, double ramp_duration, double initial_scattering_length, double final_scattering_length, std::ostream &output_stream) override

  Useful possible overload.
- void write_operator_splitting_output (size_t iteration_number, double current_scattering_length, std::ostream &output_stream) override

  Useful possible overload.

Protected Member Functions

- void solve_step_1_operator_splitting (double current_scattering_length) override

  Useful possible overload.

Additional Inherited Members

8.8.1 Member Function Documentation

8.8.1.1 run_operator_splitting()

```
void myGPSolver::run_operator_splitting (
            int ,
            double ,
            double ,
            double ,
            double ,
            std::ostream &  )  [override], [virtual]
```

Useful possible overload.

Reimplemented from UltraCold::GPSolvers::GPSolver.

8.8.1.2 solve_step_1_operator_splitting()

```
void myGPSolver::solve_step_1_operator_splitting (
            double  )  [override], [protected], [virtual]
```

Useful possible overload.

Reimplemented from UltraCold::GPSolvers::GPSolver.

8.8.1.3 write_operator_splitting_output()

```
void myGPSolver::write_operator_splitting_output (
            size_t ,
            double ,
            std::ostream &  )  [override], [virtual]
```

Useful possible overload.

Reimplemented from UltraCold::GPSolvers::GPSolver.

8.9 UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver Class Reference

Class to solve the Bogolyubov equations for a trapped Bose gas.

`#include <bogolyubov_solvers.hpp>`

Public Member Functions

- TrappedBogolyubovSolver (Vector< double > &x, Vector< double > &psi0, Vector< double > &Vext, double scattering_length, double chemical_potential, int number_of_modes, double tolerance, int maximum_↩ number_arnoldi_iterations, bool eigenvectors_required)

  Constructor for a TrappedBogolyubovSolver in one space dimension and real stationary condensate wave function.
- TrappedBogolyubovSolver (Vector< double > &x, Vector< double > &y, Vector< double > &psi0, Vector< double > &Vext, double scattering_length, double chemical_potential, int number_of_modes, double tolerance, int maximum_number_arnoldi_iterations, bool eigenvectors_required)

  Constructor for a TrappedBogolyubovSolver in two space dimensions and real stationary condensate wave function.
- TrappedBogolyubovSolver (Vector< double > &x, Vector< double > &y, Vector< double > &z, Vector< double > &psi0, Vector< double > &Vext, double scattering_length, double chemical_potential, int number↩ _of_modes, double tolerance, int maximum_number_arnoldi_iterations, bool eigenvectors_required)

  Constructor for a TrappedBogolyubovSolver in three space dimensions and real stationary condensate wave function.
- TrappedBogolyubovSolver (Vector< double > &x, Vector< std::complex< double > > &psi0, Vector< double > &Vext, double scattering_length, double chemical_potential, int number_of_modes, double tolerance, int maximum_number_arnoldi_iterations, bool eigenvectors_required)

  Constructor for a TrappedBogolyubovSolver in one space dimension and complex stationary condensate wave function.
- TrappedBogolyubovSolver (Vector< double > &x, Vector< double > &y, Vector< std::complex< double > > &psi0, Vector< double > &Vext, double scattering_length, double chemical_potential, int number_of_modes, double tolerance, int maximum_number_arnoldi_iterations, bool eigenvectors_required)

  Constructor for a TrappedBogolyubovSolver in two space dimensions and complex stationary condensate wave function.
- TrappedBogolyubovSolver (Vector< double > &x, Vector< double > &y, Vector< double > &z, Vector< std::complex< double > > &psi0, Vector< double > &Vext, double scattering_length, double chemical_potential, int number_of_modes, double tolerance, int maximum_number_arnoldi_iterations, bool eigenvectors_required)

  Constructor for a TrappedBogolyubovSolver in three space dimensions and complex stationary condensate wave function.
- std::tuple< std::vector< std::complex< double > >, std::vector< Vector< std::complex< double > > >, std::vector< Vector< std::complex< double > > > > run ()

  Solve the Bogolyubov equations.

Protected Attributes

- Vector< double > **Vext**
- Vector< double > **x**
- Vector< double > **y**
- Vector< double > **z**
- Vector< double > **kx**
- Vector< double > **ky**
- Vector< double > **kz**
- Vector< double > **kmod2**
- Vector< std::complex< double > > **temp**
- Vector< std::complex< double > > **temp2**
- Vector< std::complex< double > > **temp_tilde**
- Vector< std::complex< double > > **temp2_tilde**
- Vector< std::complex< double > > **psi0**
- std::vector< Vector< std::complex< double > > > **u**
- std::vector< Vector< std::complex< double > > > **v**
- int **nx**
- int **ny**
- int **nz**

- double **dx** = 1.0
- double **dy** = 1.0
- double **dz** = 1.0
- double **dv** = 1.0
- int **number_of_modes**
- double **chemical_potential**
- double **scattering_length**
- bool **problem_is_1d** =false
- bool **problem_is_2d** =false
- bool **problem_is_3d** =false
- bool **eigenvectors_required** =false
- bool **problem_is_real** =false
- bool **problem_is_complex** =false
- double **tolerance**
- a_int **maximum_number_of_arnoldi_iterations**

8.9.1 Detailed Description

Class to solve the Bogolyubov equations for a trapped Bose gas.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

Studying the elementary excitations of a trapped Bose gas in the condensate phase on top of a certain ground (or stationary) state, means searching for solutions of the time-dependent Gross-Pitaevskii equation of the form

$$\psi(\mathbf{r},t) = e^{-i\frac{\mu}{\hbar}t}\left[\psi_0(\mathbf{r}) + \sum_{n=0}^{\infty}\left(u_n(\mathbf{r})e^{-i\omega_n t} + v_n^*(\mathbf{r})e^{i\omega_n t}\right)\right]$$

and solving the eigenvalue problem that comes out by keeping only terms linear in the quasi-particle amplitudes u and v. In the case of the ordinary Bose gas (i.e., an ensemble of bosonic particles at very low temperature interacting only via a contact interaction), this amounts to solving the following eigenvalue problem

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\frac{\hbar^2}{2m}\nabla^2 + V_{ext}(\mathbf{r}) + g|\psi_0|^2 - \mu & g\psi_0^2 \\ -g(\psi_0^*)^2 & -\left(-\frac{\hbar^2}{2m}\nabla^2 + V_{ext}(\mathbf{r}) + g|\psi_0|^2 - \mu\right) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

In the case in which the condensate wave function is real (e.g., in absence of vortices, solitons...) the problem can be recast in a more convenient form. In fact, taking the sum and the difference between the two equations, one easily finds

$$\hat{H}\hat{X}(u+v) = (\hbar\omega)^2(u+v)$$
$$\hat{X}\hat{H}(u-v) = (\hbar\omega)^2(u-v)$$

with

$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V_{ext}(\mathbf{r}) + g|\psi_0|^2 - \mu$$
$$\hat{X} = -\frac{\hbar^2}{2m}\nabla^2 + V_{ext}(\mathbf{r}) + 3g|\psi_0|^2 - \mu$$

Now, both equations allow to find the (square) of the energy of the Bogolyubov modes, but solving a system of half the dimensionality of the original problem. This typically allows a great saving of computational time. The eigenvectors of the two problems correspond to $(u+v)$ and $(u-v)$ respectively, so that if one is interested in finding the Bogolyubov quasi-particle amplitudes u and v, one also needs to solve the second problem, and then set $u = 0.5\left((u+v)+(u-v)\right)$ and $v = 0.5\left((u+v)-(u-v)\right)$

This class solves the eigenvalue problem using the matrix-free routines provided as part of the package `arpack-ng` , which is distributed as a bundled package with UltraCold.

8.9.2 Constructor & Destructor Documentation

8.9.2.1 TrappedBogolyubovSolver() [1/6]

```
UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver::TrappedBogolyubovSolver (
            Vector< double > & x,
            Vector< double > & psi0,
            Vector< double > & Vext,
            double scattering_length,
            double chemical_potential,
            int number_of_modes,
            double tolerance,
            int maximum_number_arnoldi_iterations,
            bool eigenvectors_required )
```

Constructor for a TrappedBogolyubovSolver in one space dimension and real stationary condensate wave function.

Parameters

x	*Vector<double>* representing the cartesian axis on which the Bogolyubov equations in one space dimension will be solved
psi_0	*Vector<double>* representing a stationary condensate wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units
chemical_potential	*double* the ground-state chemical potential
eigenvectors_required	*bool* specifies if also the eigenvectors are required. Default is false.

8.9.2.2 TrappedBogolyubovSolver() [2/6]

```
UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver::TrappedBogolyubovSolver (
            Vector< double > & x,
```

```
            Vector< double > & y,
            Vector< double > & psi0,
            Vector< double > & Vext,
            double scattering_length,
            double chemical_potential,
            int number_of_modes,
            double tolerance,
            int maximum_number_arnoldi_iterations,
            bool eigenvectors_required )
```

Constructor for a TrappedBogolyubovSolver in two space dimensions and real stationary condensate wave function.

Parameters

x	*Vector<double>* representing the x-axis on which the Bogolyubov equations in one space dimension will be solved
y	*Vector<double>* representing the y-axis on which the Bogolyubov equations in one space dimension will be solved
psi_0	*Vector<double>* representing a stationary condensate wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units
chemical_potential	*double* the ground-state chemical potential
eigenvectors_required	*bool* specifies if also the eigenvectors are required. Default is false.

8.9.2.3 TrappedBogolyubovSolver() [3/6]

```
UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver::TrappedBogolyubovSolver (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & z,
            Vector< double > & psi0,
            Vector< double > & Vext,
            double scattering_length,
            double chemical_potential,
            int number_of_modes,
            double tolerance,
            int maximum_number_arnoldi_iterations,
            bool eigenvectors_required )
```

Constructor for a TrappedBogolyubovSolver in three space dimensions and real stationary condensate wave function.

Parameters

x	*Vector<double>* representing the x-axis on which the Bogolyubov equations in one space dimension will be solved
y	*Vector<double>* representing the y-axis on which the Bogolyubov equations in one space dimension will be solved
z	*Vector<double>* representing the z-axis on which the Bogolyubov equations in one space dimension will be solved
psi_0	*Vector<double>* representing a stationary condensate wave function

Parameters

Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units
chemical_potential	*double* the ground-state chemical potential
eigenvectors_required	*bool* specifies if also the eigenvectors are required. Default is false.

8.9.2.4 TrappedBogolyubovSolver() [4/6]

```
UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver::TrappedBogolyubovSolver (
            Vector< double > & x,
            Vector< std::complex< double > > & psi0,
            Vector< double > & Vext,
            double scattering_length,
            double chemical_potential,
            int number_of_modes,
            double tolerance,
            int maximum_number_arnoldi_iterations,
            bool eigenvectors_required )
```

Constructor for a TrappedBogolyubovSolver in one space dimension and complex stationary condensate wave function.

Parameters

x	*Vector<double>* representing the cartesian axis on which the Bogolyubov equations in one space dimension will be solved
psi_0	*Vector<std::complex<double>>* representing a stationary condensate wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units
chemical_potential	*double* the ground-state chemical potential
eigenvectors_required	*bool* specifies if also the eigenvectors are required. Default is false.

8.9.2.5 TrappedBogolyubovSolver() [5/6]

```
UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver::TrappedBogolyubovSolver (
            Vector< double > & x,
            Vector< double > & y,
            Vector< std::complex< double > > & psi0,
            Vector< double > & Vext,
            double scattering_length,
            double chemical_potential,
            int number_of_modes,
            double tolerance,
            int maximum_number_arnoldi_iterations,
            bool eigenvectors_required )
```

Constructor for a TrappedBogolyubovSolver in two space dimensions and complex stationary condensate wave function.

Parameters

x	*Vector<double>* representing the x-axis on which the Bogolyubov equations in one space dimension will be solved
y	*Vector<double>* representing the y-axis on which the Bogolyubov equations in one space dimension will be solved
psi_0	*Vector<std::complex<double>>* representing a stationary condensate wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units
chemical_potential	*double* the ground-state chemical potential
eigenvectors_required	*bool* specifies if also the eigenvectors are required. Default is false.

8.9.2.6 TrappedBogolyubovSolver() [6/6]

```
UltraCold::BogolyubovSolvers::TrappedBogolyubovSolver::TrappedBogolyubovSolver (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & z,
            Vector< std::complex< double > > & psi0,
            Vector< double > & Vext,
            double scattering_length,
            double chemical_potential,
            int number_of_modes,
            double tolerance,
            int maximum_number_arnoldi_iterations,
            bool eigenvectors_required )
```

Constructor for a TrappedBogolyubovSolver in three space dimensions and complex stationary condensate wave function.

Parameters

x	*Vector<double>* representing the x-axis on which the Bogolyubov equations in one space dimension will be solved
y	*Vector<double>* representing the y-axis on which the Bogolyubov equations in one space dimension will be solved
z	*Vector<double>* representing the z-axis on which the Bogolyubov equations in one space dimension will be solved
psi_0	*Vector<std::complex<double>>* representing a stationary condensate wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units
chemical_potential	*double* the ground-state chemical potential
eigenvectors_required	*bool* specifies if also the eigenvectors are required. Default is false.

8.9.3 Member Function Documentation

8.9.3.1 run()

```
std::tuple< std::vector< std::complex< double > >, std::vector< Vector< std::complex< double
> > >, std::vector< Vector< std::complex< double > > > > UltraCold::BogolyubovSolvers::↩
TrappedBogolyubovSolver::run ( )
```

Solve the Bogolyubov equations.

This member function actually solves the Bogolyubov equations for a trapped condensate. In the case in which a real wave-function was passed to the constructor, the function will use the following useful recast of the problem in one of halved dimensionality. Starting from

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -\frac{\hbar^2}{2m}\nabla^2 + V_{ext}(\mathbf{r}) + g|\psi_0|^2 - \mu & g\psi_0^2 \\ -g(\psi_0)^2 & -\left(-\frac{\hbar^2}{2m}\nabla^2 + V_{ext}(\mathbf{r}) + g|\psi_0|^2 - \mu\right) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

and taking the sum and the difference between two equations, one obtain the following equivalent formulation of the problem

$$\hat{H}\hat{X}(u+v) = (\hbar\omega)^2(u+v)$$
$$\hat{X}\hat{H}(u-v) = (\hbar\omega)^2(u-v)$$

with

$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V_{ext}(\mathbf{r}) + g|\psi_0|^2 - \mu$$
$$\hat{X} = -\frac{\hbar^2}{2m}\nabla^2 + V_{ext}(\mathbf{r}) + 3g|\psi_0|^2 - \mu$$

If the eigenvectors are not required, this function will solve only the first equation, calculating the square of the energies of the eigen-modes, but *returning the energies* (i.e., it calculates the square root before returning). If the eigenvectors are also requested, the function will solve also the second eigenvalue problem, obtaining the Bogolyubov quasi-particle amplitudes u and v by taking (half) the sum and the difference between the eigenvectors of the two problems.

In the case in which, instead, the wave-function passed to the constructor is complex, the function will solve, by brute force, the complete problem.

8.10 UltraCold::BogolyubovSolvers::TrappedDipolarBogolyubovSolver Class Reference

Class to solve the Bogolyubov equations for a trapped **dipolar** Bose gas.

```
#include <bogolyubov_solvers.hpp>
```

Public Member Functions

- **TrappedDipolarBogolyubovSolver** (Vector< double > &x, Vector< double > &y, Vector< double > &psi0, Vector< double > &Vext, double scattering_length, double dipolar_length, double chemical_potential, int number_of_modes, double tolerance, int maximum_number_arnoldi_iterations, bool eigenvectors_required)
- TrappedDipolarBogolyubovSolver (Vector< double > &x, Vector< double > &y, Vector< double > &z, Vector< double > &psi0, Vector< double > &Vext, double scattering_length, double dipolar_length, double chemical_potential, int number_of_modes, double tolerance, int maximum_number_arnoldi_iterations, bool eigenvectors_required)

  Constructor for a TrappedDipolarBogolyubovSolver in three space dimensions and real stationary condensate wave function.
- std::tuple< std::vector< std::complex< double > >, std::vector< Vector< std::complex< double > > >, std::vector< Vector< std::complex< double > > > > run ()

  Solve the Bogolyubov equations.

Protected Attributes

- Vector< double > **Vext**
- Vector< double > **x**
- Vector< double > **y**
- Vector< double > **z**
- Vector< double > **kx**
- Vector< double > **ky**
- Vector< double > **kz**
- Vector< double > **kmod2**
- Vector< std::complex< double > > **temp**
- Vector< std::complex< double > > **temp2**
- Vector< std::complex< double > > **temp_tilde**
- Vector< std::complex< double > > **temp2_tilde**
- Vector< std::complex< double > > **psi0**
- Vector< std::complex< double > > **Vtilde**
- Vector< std::complex< double > > **Phi_dd**
- std::vector< Vector< std::complex< double > > > **u**
- std::vector< Vector< std::complex< double > > > **v**
- int **nx**
- int **ny**
- int **nz**
- double **dx** = 1.0
- double **dy** = 1.0
- double **dz** = 1.0
- double **dv** = 1.0
- int **number_of_modes**
- double **chemical_potential**
- double **scattering_length**
- double **epsilon_dd**
- double **gamma_epsilon_dd**
- bool **eigenvectors_required** =false
- bool **problem_is_real** =false
- bool **problem_is_complex** =false
- double **tolerance**
- a_int **maximum_number_of_arnoldi_iterations**

8.10.1 Detailed Description

Class to solve the Bogolyubov equations for a trapped **dipolar** Bose gas.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

Studying the elementary excitations of a trapped **dipolar** Bose gas in the condensate phase on top of a certain ground (or stationary) state, means searching for solutions of the time-dependent extended Gross-Pitaevskii equation of the form

$$\psi(\mathbf{r},t) = e^{-i\frac{\mu}{\hbar}t}\left[\psi_0(\mathbf{r}) + \sum_{n=0}^{\infty}\left(u_n(\mathbf{r})e^{-i\omega_n t} + v_n^*(\mathbf{r})e^{i\omega_n t}\right)\right]$$

and solving the eigenvalue problem that comes out by keeping only terms linear in the quasi-particle amplitudes u and v. In the case of a **dipolar** Bose gas, taking also into account the effects of quantum fluctuations via the Lee-Huang-Yang (LHY) correction, this amounts to solving the following eigenvalue problem

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \hat{H} - \mu + \hat{X} & \hat{X}^\dagger \\ -\hat{X} & -(\hat{H} - \mu + \hat{X}^\dagger) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

with

$$\begin{aligned}\mathcal{H}(\mathbf{r}) = &-\frac{\hbar^2}{2m}\nabla^2 + V_{\text{ext}}(\mathbf{r}) + g|\Psi(\mathbf{r},t)|^2 + \gamma(\varepsilon_{dd})|\Psi(\mathbf{r},t)|^3 \\ &+ \int d\mathbf{r}' V_{dd}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}',t)|^2\,,\end{aligned}$$

and

f(**r**) = $\psi_0(\mathbf{r})\int d\mathbf{r}' V_{dd}(\mathbf{r}-\mathbf{r}')f(\mathbf{r}')\psi_0^*(\mathbf{r}') + \frac{3}{2}\gamma(\varepsilon_{dd})|\psi_0(\mathbf{r})|^3 f(\mathbf{r})$

and finally

$$\gamma(\varepsilon_{dd}) = \frac{16}{3\sqrt{\pi}} g a^{\frac{3}{2}}\,\text{Re}\left[\int_0^\pi d\theta \sin\theta[1+\varepsilon_{dd}(3\cos^2\theta - 1)]^{\frac{5}{2}}\right].$$

with $g = 4\pi\hbar^2 a/m$ the coupling constant fixed by the s-wave scattering length a, $V_{dd}(\mathbf{r}_i - \mathbf{r}_j) = \frac{\mu_0\mu^2}{4\pi}\frac{1-3\cos^2\theta}{|\mathbf{r}_i-\mathbf{r}_j|^3}$ the dipole-dipole potential, being μ_0 the magnetic permeability in vacuum, μ the magnetic dipole moment and θ the

angle between the vector distance between dipoles and the polarization direction, which we choose as the x-axis, and $\varepsilon_{dd} = \mu_0\mu^2/(3g) = a_{dd}/a$ the ratio between the strength of the dipolar and the contact interaction, eventually written in terms of the dipolar length a_{dd} and the scattering length a.

In the case in which the condensate wave function is real (e.g., in absence of vortices, solitons...) the problem can be recast in a more convenient form. In fact, taking the sum and the difference between the two equations, one easily finds

$$(\hat{H} - \mu)(\hat{H} - \mu + 2\hat{X})(u + v) = (\hbar\omega)^2(u + v)$$
$$(\hat{H} - \mu + 2\hat{X})(\hat{H} - \mu)(u - v) = (\hbar\omega)^2(u - v)$$

Now, both equations allow to find the (square) of the energy of the Bogolyubov modes, but solving a system of half the dimensionality of the original problem. This typically allows a great saving of computational time. The eigenvectors of the two problems correspond to $(u + v)$ and $(u - v)$ respectively, so that if one is interested in finding the Bogolyubov quasi-particle amplitudes u and v, one also needs to solve the second problem, and then set $u = 0.5\,((u + v) + (u - v))$ and $v = 0.5\,((u + v) - (u - v))$

This class solves the eigenvalue problem using the matrix-free routines provided as part of the package `arpack-ng` , which is distributed as a bundled package with UltraCold.

8.10.2 Constructor & Destructor Documentation

8.10.2.1 TrappedDipolarBogolyubovSolver()

```
UltraCold::BogolyubovSolvers::TrappedDipolarBogolyubovSolver::TrappedDipolarBogolyubovSolver (
            Vector< double > & x,
            Vector< double > & y,
            Vector< double > & z,
            Vector< double > & psi0,
            Vector< double > & Vext,
            double scattering_length,
            double dipolar_length,
            double chemical_potential,
            int number_of_modes,
            double tolerance,
            int maximum_number_arnoldi_iterations,
            bool eigenvectors_required )
```

Constructor for a TrappedDipolarBogolyubovSolver in three space dimensions and real stationary condensate wave function.

Parameters

x	*Vector<double>* representing the x-axis on which the Bogolyubov equations will be solved
y	*Vector<double>* representing the y-axis on which the Bogolyubov equations will be solved
z	*Vector<double>* representing the z-axis on which the Bogolyubov equations will be solved
psi_0	*Vector<double>* representing a stationary condensate wave function
Vext	*Vector<double>* representing the external potential.
scattering_length	*double* the scattering length in appropriate units
dipolar_length	*double* the dipolar length in appropriate units
chemical_potential	*double* the ground-state chemical potential
eigenvectors_required	*bool* specifies if also the eigenvectors are required. Default is false.

8.10.3 Member Function Documentation

8.10.3.1 run()

`std::tuple< std::vector< std::complex< double > >, std::vector< Vector< std::complex< double > > >, std::vector< Vector< std::complex< double > > > > UltraCold::BogolyubovSolvers::TrappedDipolarBogolyubovSolver::run ( )`

Solve the Bogolyubov equations.

This member function actually solves the Bogolyubov equations for a trapped dipolar condensate. In the case in which a real wave-function was passed to the constructor, the function will use the following useful recast of the problem in one of halved dimensionality. Starting from

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \hat{H} - \mu + \hat{X} & \hat{X}^\dagger \\ -\hat{X} & -(\hat{H} - \mu + \hat{X}^\dagger) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

and taking the sum and the difference between two equations, one obtain the following equivalent formulation of the problem

$$(\hat{H} - \mu)(\hat{H} - \mu + 2\hat{X})(u + v) = (\hbar\omega)^2(u + v)$$
$$(\hat{H} - \mu + 2\hat{X})(\hat{H} - \mu)(u - v) = (\hbar\omega)^2(u - v)$$

with

$$\mathcal{H}(\mathbf{r}) = -\frac{\hbar^2}{2m}\nabla^2 + V_{\text{ext}}(\mathbf{r}) + g|\Psi(\mathbf{r},t)|^2 + \gamma(\varepsilon_{dd})|\Psi(\mathbf{r},t)|^3 + \int d\mathbf{r}' V_{dd}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}',t)|^2\,,$$

and

f(**r**) = $\psi_0(\mathbf{r}) \int d\mathbf{r}' V_{dd}(\mathbf{r}-\mathbf{r}') f(\mathbf{r}')\psi_0^*(\mathbf{r}') + \frac{3}{2}\gamma(\varepsilon_{dd})|\psi_0(\mathbf{r})|^3 f(\mathbf{r})$

and finally

$$\gamma(\varepsilon_{dd}) = \frac{16}{3\sqrt{\pi}} g a^{\frac{3}{2}}\, \text{Re}\left[\int_0^\pi d\theta \sin\theta [1 + \varepsilon_{dd}(3\cos^2\theta - 1)]^{\frac{5}{2}}\right].$$

with $g = 4\pi\hbar^2 a/m$ the coupling constant fixed by the s-wave scattering length a, $V_{dd}(\mathbf{r}_i - \mathbf{r}_j) = \frac{\mu_0\mu^2}{4\pi}\frac{1-3\cos^2\theta}{|\mathbf{r}_i-\mathbf{r}_j|^3}$ the dipole-dipole potential, being μ_0 the magnetic permeability in vacuum, μ the magnetic dipole moment and θ the angle between the vector distance between dipoles and the polarization direction, which we choose as the x-axis, and $\varepsilon_{dd} = \mu_0\mu^2/(3g) = a_{dd}/a$ the ratio between the strength of the dipolar and the contact interaction, eventually written in terms of the dipolar length a_{dd} and the scattering length a.

If the eigenvectors are not required, this function will solve only the first equation, calculating the square of the energies of the eigen-modes, but *returning the energies* (i.e., it calculates the square root before returning). If the eigenvectors are also requested, the function will solve also the second eigenvalue problem, obtaining the Bogolyubov quasi-particle amplitudes u and v by taking (half) the sum and the difference between the eigenvectors of the two problems.

In the case in which, instead, the wave-function passed to the constructor is complex, the function will solve, by brute force, the complete problem.

8.11 UltraCold::Vector< T > Class Template Reference

A class that represents arrays of numerical elements.

`#include <vector.hpp>`

Public Member Functions

- Vector (int)
  Constructor for a one-dimensional array.
- Vector (int, int)
  Constructor for a two-dimensional array.
- Vector (int, int, int)
  Constructor for a two-dimensional array.
- **Vector** (const Vector< T > &)
  Copy constructor.
- Vector & **operator=** (const Vector< T > &)
  Copy assignment.
- **Vector** (Vector< T > &&)
  Move constructor.
- Vector & **operator=** (Vector< T > &&)
  Move assignment.
- ~**Vector** ()
  Destructor, releases the memory allocated by mkl_calloc.
- void **reinit** (int)
  Reinitialize a one-dimensional Vector.
- void **reinit** (int, int)
  Reinitialize a two-dimensional Vector.
- void **reinit** (int, int, int)
  Reinitialize a three-dimensional Vector.
- int **size** ()
  Get the total number of elements.
- int **order** ()
  Get the total number of dimensions.
- int extent (int)

Get the extent of the Vector along a certain direction.

- T * data ()

  Get the pointer to the first array element.

- T & operator() (int)

  Member access operators in Fortran style for a one-dimensional Vector.

- T & operator() (int, int)

  Member access operators in Fortran style for a two-dimensional Vector.

- T & operator() (int, int, int)

  Member access operators in Fortran style for a three-dimensional Vector.

- T & operator[ ] (int)

  Plain and simple member access operators in C-style.

8.11.1 Detailed Description

template<typename T>
class UltraCold::Vector< T >

A class that represents arrays of numerical elements.

Author

Santo Maria Roccuzzo (santom.roccuzzo@gmail.com)

The purpose of this class is to provide a useful way to store arrays of numerical elements in contiguous memory spaces and to use optimized mathematical operations acting on them.
These include:

- Optimized memory allocation through Intel's MKL memry functions,
- Fourier transforms (both forward and backward),
- Fortran-style access to the array elements

Note

Explicit instantiations are provided only for types types `double` and `std::complex<double>`, which is equivalent to the C complex type defined in `complex.h`.

8.11.2 Constructor & Destructor Documentation

8.11.2.1 Vector() [1/3]

```
template<typename T >
UltraCold::Vector< T >::Vector (
            int n1 )
```

Constructor for a one-dimensional array.

Parameters

n1	*int* The extent of the array

8.11.2.2 Vector() [2/3]

```
template<typename T >
UltraCold::Vector< T >::Vector (
            int n1,
            int n2 )
```

Constructor for a two-dimensional array.

Parameters

n1	*int* The extent of the array along the first dimension
n2	*int* The extent of the array along the second dimension

8.11.2.3 Vector() [3/3]

```
template<typename T >
UltraCold::Vector< T >::Vector (
            int n1,
            int n2,
            int n3 )
```

Constructor for a two-dimensional array.

Parameters

n1	*int* The extent of the array along the first dimension
n2	*int* The extent of the array along the second dimension
n3	*int* The extent of the array along the third dimension

8.11.3 Member Function Documentation

8.11.3.1 data()

```
template<typename T >
T * UltraCold::Vector< T >::data
```

Get the pointer to the first array element.

Use it with care!

8.11.3.2 extent()

```
template<typename T >
int UltraCold::Vector< T >::extent (
            int i )
```

Get the extent of the Vector along a certain direction.

If a wrong index is given, returns -1.

Parameters

i	*int* direction along which we want to get the extent

8.11.3.3 operator()() [1/3]

```
template<typename T >
T & UltraCold::Vector< T >::operator() (
            int i )
```

Member access operators in Fortran style for a one-dimensional Vector.

Parameters

i	*int* the index of the requested element

8.11.3.4 operator()() [2/3]

```
template<typename T >
T & UltraCold::Vector< T >::operator() (
            int i,
            int j )
```

Member access operators in Fortran style for a two-dimensional Vector.

Parameters

i	*int* first index of the requested element
j	*int* second index of the requested element

Note

Only the *syntax* is Fortran-style, while the elements are accessed in C-style row-major order.

8.11.3.5 operator()() [3/3]

```
template<typename T >
T & UltraCold::Vector< T >::operator() (
            int i,
            int j,
            int k )
```

Member access operators in Fortran style for a three-dimensional Vector.

Parameters

i	*int* first index of the requested element
j	*int* second index of the requested element
k	*int* third index of the requested element

Note

Only the *syntax* is Fortran-style, while the elements are accessed in C-style row-major order.

8.11.3.6 operator[]()

```
template<typename T >
T & UltraCold::Vector< T >::operator[] (
            int i )
```

Plain and simple member access operators in C-style.

Parameters

i	*int* the index of the requested element