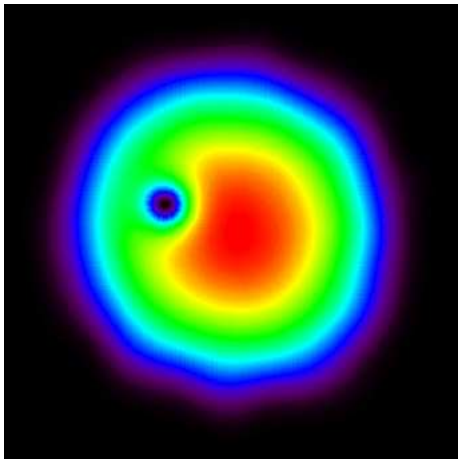
The GPE Code

Anthony J. Youd



School of Mathematics & Statistics
Newcastle University
Newcastle upon Tyne
United Kingdom

Wed 27 Nov 14:23:53 2013

Contents

Chapter 1

Introduction

The GPE code is a modular 3D Gross–Pitaevskii equation solver, written in Fortran 90 and parallelised using the Message Passing Interface (MPI).

The code can solve a range of problems for both homogeneous and non-homogeneous (trapped) condensates, including vortex dynamics (lines, rings), and non-equilibrium dynamics (condensate formation).

Time stepping is performed explicitly using one of the following methods:

- first-order Euler (E1);
- second-order Runge–Kutta (RK2);
- fourth-order Runge–Kutta (RK4);
- fourth/fifth-order adaptive Runge–Kutta–Fehlberg (RK45).

The spatial discretisation is performed with either second- or fourth-order accurate centred finite differences, and the boundary conditions can be either periodic or reflective.

The "kind" of floating point variables is parametrised, so real or double-precision arithmetic can be specified without relying on compiler switches.

The code is released under the Apache 2.0 licence, and while this permits anyone to alter the code in any way they see fit, I should be grateful if you would acknowledge me in publications that have used the code or parts of it.

Some publications that have used the code include Al-Amri *et al.* (2008), White *et al.* (2010), Tebbs *et al.* (2011), and Helm *et al.* (2011).

1.1 Preliminaries and required software

Since the code is parallelised using MPI, an MPI parallel environment must be available (currently, this is also true even when running on one processor). The code has been tested with the MPICH and OpenMPI implementations of MPI, but other implementations which adhere to the MPI standard should also be useable.

The code uses some Fortran 2003 features, including allocatable arrays within user-defined types, the `protected` keyword, and stream I/O, so a compiler which allows some Fortran 2003 constructs is required. Compilers known to work include sunf95 (Sun Studio 12), ifort (Intel Fortran Compiler), and gfortran (GNU Fortran Compiler).

The code also requires the FFTw library (`http://www.fftw.org`) for some routines. This must be version 2 of the library (the latest is version 2.1.5), since FFTs in the code are performed in parallel, and FFTw version 3 does not yet support (distributed) parallel transforms.

To view the output of the code, any graphics program capable of reading space-separated, columnar text files will be able to produce time-series plots (gnuplot is a nice, easy to use program, `http://www.gnuplot.info/`). Routines written in IDL (Interactive Data Language, `http://www.ittvis.com/ProductServices/IDL.aspx`) for producing 2D contour and 3D isosurface plots are included with the code. The data for these plots are saved in binary format, so more work will be needed if a different graphics program is to be used. Volume renderings using VAPOR (`http://www.vapor.ucar.edu/`) are also possible.

1.1.1 Summary of required hardware and software

The following outlines the required hardware and software needed to run the code. Other hardware and software may work, but has not been tested.

- A Unix-like operating system, running on x86 or x86-64 hardware.
- A standard development environment including make.
- A Fortran 90 compiler, supporting Fortran 2003 constructs (see above).
- An implementation of MPI, e.g. OpenMPI or MPICH.
- Version 2.x.y of the FFTw library.
- gnuplot and/or IDL for visualisation; VAPOR optional.

1.1.2 IDL

If you intend to use IDL to produce 2D contour and 3D isosurface plots, then you will need to make sure that some environment variables are set in your startup files, e.g. `.cshrc` if you are using C Shell, or `.profile` if you are using Bash. These variables are:

- `IDL_DIR` — specifies the IDL install directory;
- `IDL_PATH` — specifies the directories in which IDL will look to find IDL procedures and functions;

- `IDL_STARTUP` — specifies where to find the IDL startup file `.idlrc`.

Two example files in the `idl` subdirectory of the main code directory, named `idl.csh` and `idl.sh` (for C Shell and Bash respectively), provide example commands to set these environment variables. They will likely need to be edited to suit your system, then added to your startup file.

An example `.idlrc` file is also provided (named `dot.idlrc`), which fixes some quirks with X displays, and sets 24-bit true colour output by default. If you wish to use this, then make sure that the `IDL_STARTUP` environment variable points to this file.

IDL command line

You might find that IDL's command line is broken, i.e. no command-line completion, no command-line history, inability to use the `End` or `Home` keys, etc. To remedy this, install the `rlwrap` package, and set up an alias `alias -a -c idl` for the `idl` command. This will provide `readline` functionality for IDL.

1.2 Organisation of this manual

The rest of this manual is organised as follows:

- Chapter 2 is a getting started guide. The chapter will guide you through the basics of setting up, compiling, and running the code, as well as visualising some of the results.
- Chapter 3 describes the governing equations, and the non-dimensionalisation used.
- Chapter 4 describes some of the numerical formulation, including time stepping schemes, spatial discretisation, and boundary conditions.
- Chapter 5 describes the files which make up the code, including source files, input and output files, and the IDL programs used for visualisation.
- Appendix A gives detailed derivations of the various non-dimensionalised equations, and other variables and quantities.

Chapter 2

Getting started

This chapter is intended as a quick start guide to help you compile, run, and view the results of the GPE code. Later chapters explain various aspects of the code in greater detail, and some amount of reading the (commented) source code should be expected, to gain a better understanding of what the code can do. Here, we shall outline the basics of going through a run cycle, by using the provided *ring* example, which propagates a vortex ring in a homogeneous condensate for a short time.

2.1 Makefile

Change to the `src` directory, and copy `Makefile.dist` to `Makefile`. The settings in the Makefile must be correct, in order for the code to compile. The default Makefile assumes that compilation will be with the GNU Fortran compiler (gfortran), and that 64-bit code should be generated. If this is not what you want, then you will need to make changes. See §5.3 for more information.

2.2 Setting the run parameters and initial condition

Change to the `examples/ring` directory. In here, you will find symbolic links to the source files, and in addition, three `.in.dist` files. Before doing anything else, copy these files to files with the same name, but with the `.dist` extension removed. In general, it is these files — `parameters.in`, `ic.in`, and `run.in` — that you will need to edit, in order to set up a run.

2.2.1 parameters.in

For this example, simply change `nyprocs` and `nzprocs`, so that `nyprocs*nzprocs` is equal to (or less than) the number of processors on your machine. If memory is

an issue, then you may want to reduce any or all of `nx`, `ny`, or `nz` (and possibly `xr`, `yr`, and `zr` below). See §5.2.1 for a full description of this file.

2.2.2 ic.in

This file defines the initial condition. For this example, no changes are necessary. See §5.2.2 for a full description of this file.

2.2.3 run.in

This file defines the main parameters for the run, as a set of Fortran `namelist`s.

For the *ring* example, you might want to alter `end_time` if the run takes too long. If you altered any of `nx`, `ny`, or `nz` in `parameters.in`, then you might also want to alter `xr`, `yr`, or `zr` as appropriate. These parameters set the right-hand end of the physical extent of the computational box (the left-hand end is set to the same value with opposite sign). See §5.2.3 for a full description of this file.

2.3 Compiling the code

Choose a directory where your run will take place (we shall assume the directory `run_dir` in what follows). This directory should not be on a network file system (NFS), which will likely be too slow, and possibly under quota restrictions (as might be the case for a network mounted `/home` area, for example). The *ring* example will require approximately 3GB of disk space with the default parameters.

Now compile the code with

```
./setup run_dir
```

This will compile the code, create the directory `run_dir`, and copy `parameters.in`, `ic.in`, `run.in`, and `run.sh` to the run directory. In addition, the executable `gpe` is moved to the run directory.

This run uses double-precision floating-point arithmetic. For other runs, if you require only single precision, then remember to set `pr` appropriately in `parameters.in`, and compile with

```
./setup run_dir single
```

Now change to the run directory, ready to run the code.

2.4 Running the code

To run the code simply type

```
./run.sh <nprocs>
```

where <nprocs> specifies on how many processes the code should run. Ideally, this should be less than or equal to the number of physical processors in your system. It should match nyprocs*nzprocs in parameters.in, otherwise an error will result. The job is run in the background using nohup, so control immediately returns to the terminal, and logging out of the machine on which the job is running will not terminate the job.

The run.sh script has several options for controlling how the run should be started. Run it with no arguments to see usage instructions.

2.5 During the run

As a first sanity check that the job is running, look for a file called ERROR in the run directory. If this does not exist, then that is a good sign. Also have a look at the file log.txt; this should say something like

```
Explicit fifth order Runge-Kutta-Fehlberg adaptive time stepping
Homogeneous condensate, natural units non-dim.
-2i*dpsi/dt + 2iU*dpsi/dx = del^2(psi) + (1-|psi|^2)psi
```

and is something to check to make sure the correct time stepping scheme is being used, and that the correct form of the GPE is being solved.

The command top should show the gpe executable listed the same number of times as the number of processes on which you chose to run the job.

Finally, if numbers are appearing in the *.dat files, then you can be sure that the job is running.

2.6 Important run-time files

This section describes some important files which may exist in the run directory, as a job is running.

2.6.1 The RUNNING file

When the job is running, there is an empty file called RUNNING in the run directory; when the job is finished this file is deleted. You can also delete this file at any time to immediately stop the run cleanly. The job should never be ended by using kill or killall, unless an unrecoverable error occurs.

2.6.2 The ERROR file

If a serious error occurs, which the code can anticipate and handle, then the ERROR file is created in the run directory. Its contents will provide details of the error. The job will be terminated cleanly in this case, and the RUNNING file will be removed.

2.6.3 The log.txt file

Any output that the job would normally print to the screen, is instead redirected to the file `log.txt`. It is a good idea to periodically check this file to make sure that no unexpected run-time errors have occurred. Any errors which are written to `ERROR` are also written to this file. Any errors which the code could not anticipate, such as MPI communication errors, will appear here. In this case the job will most likely not terminate cleanly, and the `RUNNING` file might not be removed.

2.6.4 The SAVE file

As the job is running, data are periodically saved, according to the `save_rate` parameters in `run.in`. Every time step, the code checks for the existence of a file called `SAVE` in the run directory. If this file exists, then 3D data (from which isosurfaces and contour plots can be produced) is immediately written to disk. The `SAVE` file is then automatically removed. This file can be created, for example, by using the `touch` command, i.e. `touch SAVE`.

2.7 Program output

Once a job has started, the code produces various output files containing various data. Exactly which files appear depends on which parameters have been set in the `io_params` namelist in `run.in`.

The numbered `proc` directories contain data local to each process on which the job is running. This is where 3D isosurface data are stored. A full description of the output files is given in §5.4.

2.8 Viewing results

During and after a run, it is possible to plot various 1D graphs, 2D contour plots and 3D isosurfaces. For the 1D time-series graphs, gnuplot is a good program to use, although nearly any plotting program capable of understanding simple columnar text files will be sufficient. This section will describe how to produce some plots.

2.8.1 1D time-series plots

1D time-series data is produced in the `*.dat` files at the top-level of the run directory, much of which is self-explanatory, given the names of the files, e.g. `energy.dat`, `mass.dat`. A full description of what can be found in these data files is given in §5.4, but note that the exact contents of files can be subject to change.

So, for the *ring* example, using gnuplot to plot the total energy in the system, you could do

```
p "energy.dat" u 1:3 w lp
```

which will plot column three (energy) versus column one (real time) in the file, and where the `plot`, `using`, `with`, and `linespoints` keywords have been abbreviated to the shortest non-ambiguous form, which is possible with any gnuplot command.

As it turns out, there are not many interesting 1D graphs for the *ring* example!

2.8.2 Contour plots

Contour and isosurface data are stored in binary form within the `proc` directories; the layout of the binary files is described in table 5.8 in §5.4.1.

If you have access to the Interactive Data Language (IDL) graphics program, then IDL routines are provided with the code to directly plot the data. These are located in the `idl` subdirectory of the main code directory. The main program is called `gpe.pro`, and is described in more detail in §5.6. The subsequent sections will guide you through the basics of using the program.

From the run directory, start IDL with the command `idl`. Providing the IDL environment is set up correctly (see §1.1.2) you should be able to type

```
gpe, 0, 0, /dbl, /cntr
```

which will show a contour plot of the density of the ring initial condition as a slice through the (x, y)-plane at $z = 0$. You should see something which looks like figure 2.1.

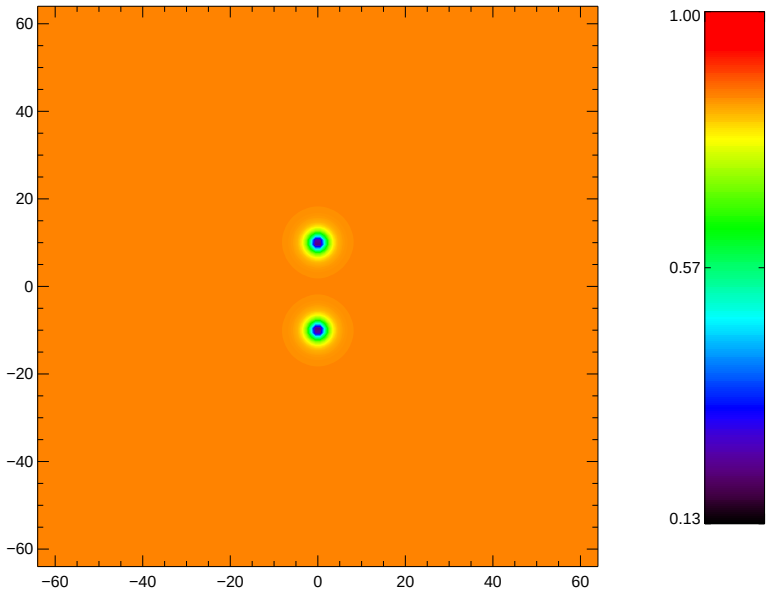


Figure 2.1: Contour plot of the density $|\psi|^2$ of the condensate, showing a vortex ring of radius $R_0 = 10$ in the (x, y)-plane at $z = 0$.

The word `gpe` represents the IDL program, which takes two non-optional arguments. The first is the index corresponding to the first file you want to plot, and the second is the index corresponding to the last file you want to plot. For output to the screen, these numbers should always be the same.

The numbers refer to the files within the `proc` directories. If you look in `proc00`, for example, you will see a number of files named `dens*******.dat`. The arguments to `gpe` are the numbers within the filenames, excluding the leading zeroes.

So for example, arguments of 0 and 0 would plot `dens0000000.dat`; arguments of 334 and 334 would plot `dens0000334.dat`, etc.

The remaining arguments are optional. The first, `/dbl`, denotes that the binary data are double-precision. You must always provide this keyword when you have performed a double-precision run. In a single-precision run, this keyword is not necessary.

The second, `/cntr`, is a keyword which turns on contour output, rather than the default, which is an isosurface (see §2.8.3).

2.8.3 Isosurface plots

Now do exactly as in §2.8.2, but this time leave off the `/cntr` keyword.

You should see an isosurface plot in a window similar to that in figure 2.2, and a control panel with various buttons on it. Most of the control panel buttons can be ignored. The `bbox`, `content`, and `axis` buttons turn on or off the bounding box, the content, and the axes labels respectively. You will need to click the `Redraw` button if you make any changes. Left-click and hold in the white window and you can move the view around; right-clicking zooms in and out, and middle-clicking moves the centre viewpoint. Depending on the speed of your machine, and the complexity of the displayed image, zooming and moving the view might be a little slow, so move the mouse slowly, and avoid large, jerky movements.

The coloured bar in the control panel is a histogram of the density, and clicking in here will redraw the isosurface at the new density level, which is shown toward the right-hand side of the control panel.

Pressing `auto` will automatically redraw the isosurface when other boxes are checked, for example, the axes, or whether the surface is solid, wireframe, or a series of points. If the display is too slow, selecting `points` or `wireframe` instead of `solid` will speed it up.

The TMat

The TMat, or transformation matrix, controls the isosurface view, and is set in `gpe.pro` to give a perspective view by default. If you would like to change the default, find the view you would like by rotating and zooming the display, then click the `TMat` button on the control panel. This will print the transformation matrix for the view in the terminal. Copy and paste this into `gpe.pro`, overwriting the TMat that is currently defined.

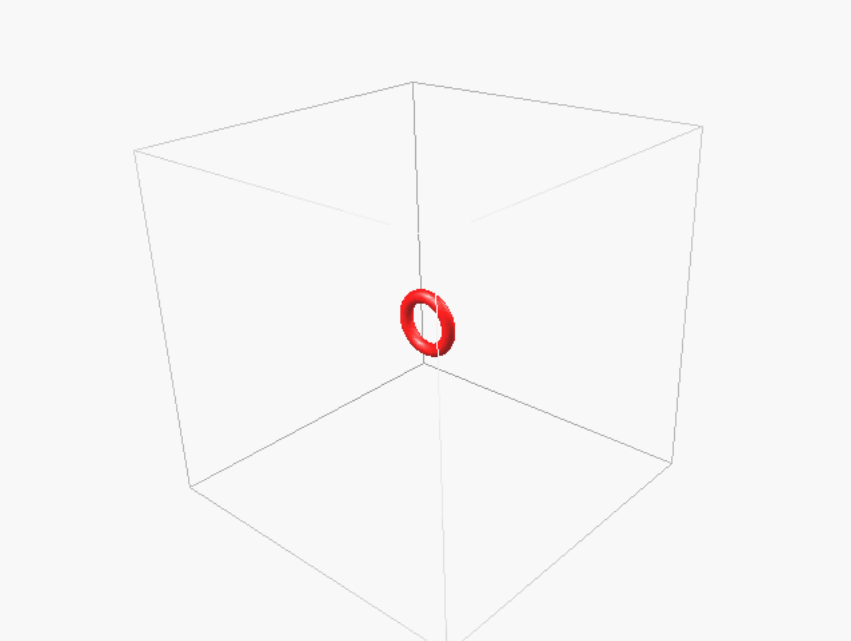


Figure 2.2: Isosurface plot of the density of the condensate at the level $|\psi|^2 = 0.75$, showing a vortex ring of radius $R_0 = 10$.

2.8.4 Animations

By saving a series of snapshots of 2D or 3D data, it is possible to then combine them into an animation.

From the run directory, run the script `create-links` with the argument `links`. This will create a directory `links`, which contains renamed symbolic links (shortcuts) pointing to the isosurface files within each process directory. This is to make sure that the files are numbered sequentially. The `create-links` script is located in the `scripts` subdirectory of the main code directory. Running it with no arguments, or with the options `-h` or `--help`, will provide usage instructions.

Contour animations

Change to the links directory and create another directory called `images`, then start IDL again.

Now type

```
gpe, 0, 9, /dbl, /cntr, /c_anim
```

This will loop over all data files numbered 0 to 9 and save `con_dens*******.png` files in the `images` directory. (If you left the *ring* example parameters at their defaults, then you should actually find 100 data files in each `proc` directory, so you could do plots from 0 to 99 if you wish.)

If you forget the `/c_anim` keyword here, you will get all the plots shown to the screen. When you are plotting 100 or more files, this can be very bad and in some cases might run your machine out of memory.

Now change to the `images` directory and run the `makemovie` script, which is located in the `scripts` subdirectory of the main code directory:

```
makemovie -i png -p con_dens
```

which will create an AVI animation out of the PNG files, with 10 frames, saving the output as `output.avi`. The `makemovie` script has several options. Run the script with no arguments to see its help.

You can then play the animation with any media player, for example,

```
mplayer output.avi
```

The animation is very short; if it works you can try a longer animation by repeating the `gpe` IDL command with arguments 0 and 99.

Isosurface animations

Rename the `images` directory to something else, and recreate an empty `images` directory. From within IDL type

```
gpe, 0, 9, /png
```

As for the contour plots before, this will save PNG files in the `images` directory, which you can then convert into an animation using `makemovie`.

If you forget the `/png` keyword, all the output will again go to the screen. This should be avoided if possible.

2.9 Restarting the code

After a run, you might decide that you would like to restart the code from where you left off, for example, maybe you started your run in imaginary time, and now want to switch to real time, or maybe you did not run for quite long enough.

If this is the case, create a new directory where the restarted run will take place, and copy `parameters.in`, `ic.in`, `run.in`, `run.sh`, and the executable `gpe` from the initial run directory to the restart directory.

Now edit `run.in`, change `restart` to `.true.`, and make any other changes you feel necessary. Recompilation is not needed when making changes only to `run.in`.

Now type

```
./run.sh -r <initial run directory> <nprocs>
```

where `<initial run directory>` is the directory of the run that you want to continue. This will copy the `end_state.dat` files from the `proc` directories to the restart directory (suitably renamed), and start the run. The `log.txt` file should then report that the code is

```
Getting restart conditions
```

and will restart from where it left off.

2.9.1 Combining a restarted run with another initial condition

Since an initial condition can be constructed by multiplying any valid wave functions together, it is possible to multiply the wave function as it was when the previous run ended with the initial condition defined in `ic.in`. To do this, perform a restart as explained above, while also setting `multiply_ic_restart` to `.true.`.

Chapter 3

Governing equations and non-dimensionalisation

The single-particle complex wavefunction $\psi(\boldsymbol{r}, t)$ for N bosons of mass m, obeys the 3-dimensional, time dependent, Gross–Pitaevskii (GP) equation (Gross, 1961; Pitaevskii, 1961)

$$\mathrm{i}\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2\psi + V_{\mathrm{ext}}\psi + g|\psi|^2\psi - \mu\psi, \tag{3.1}$$

where $\hbar = h/(2\pi)$ is the reduced Planck constant, V_{ext} is an external trapping potential, g is the strength of the interactions between the bosons, and μ is the chemical potential. The wavefunction is normalised by the condition that

$$\int_V |\psi|^2 \mathrm{d}V = N.$$

The GPE describes the evolution of the ground state of a quantum system of weakly interacting bosons, in the limit of zero temperature, and when the number of bosons N is large.

3.1 Imaginary time

In making the transformation $t \to -\mathrm{i}t$, equation (3.1) becomes

$$-\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2\psi + V_{\mathrm{ext}}\psi + g|\psi|^2\psi - \mu\psi,$$

which can be thought of as a modified diffusion equation. For certain initial conditions, it is sometimes necessary to run the code in imaginary time, because the initial conditions are not exact solutions of the GPE. By propagating in imaginary time, these solutions tend to the ground state, which is an exact solution of the GPE. The code can then be propagated in real time.

To propagate in imaginary time set `real_time` equal to `.true.`; otherwise set `real_time` equal to `.false.`.

3.2 Non-dimensionalised GPE

The code solves a non-dimensionalised form of equation (3.1). There are two possibilities for this non-dimensionalisation, depending on whether the external trapping potential is present. Here, we show the two cases.

3.2.1 Natural units

When the external trapping potential is absent, natural units are used to non-dimensionalise the GPE. In this case the scalings are

$$\begin{aligned} t &\to \frac{\hbar}{2\mu} t, \\ \boldsymbol{r} &\to a\boldsymbol{r}, \\ \psi &\to \psi_\infty \psi, \end{aligned} \tag{3.2}$$

where $a = \hbar/(\sqrt{2m\mu})$ is the healing length, and $\psi_\infty = \sqrt{\mu/g}$ is the bulk value of ψ.

These scalings lead to the dimensionless form of the GPE

$$-2\mathrm{i}\frac{\partial \psi}{\partial t} = \nabla^2\psi + \left(1 - |\psi|^2\right)\psi.$$

See appendix A for a full derivation. To solve this form of the GPE, set `eqn_to_solve` to `1` in `run.in`.

3.2.2 Harmonic oscillator units

When the external trapping potential is present, harmonic oscillator units are used to non-dimensionalise the GPE. The scalings are

$$\begin{aligned} t &\to \frac{t}{\overline{\omega}}, \\ \boldsymbol{r} &\to a_{\mathrm{OH}}\boldsymbol{r}, \\ \psi &\to a_{\mathrm{OH}}^{-\frac{3}{2}}\psi, \\ g &\to a_{\mathrm{OH}}^3 \hbar\overline{\omega} g, \\ \mu &\to \hbar\overline{\omega}\mu, \\ V_{\mathrm{ext}} &\to \hbar\overline{\omega} V_{\mathrm{ext}}, \end{aligned} \tag{3.3}$$

where $\overline{\omega} = (\omega_x\omega_y\omega_z)^{1/3}$, ω_i is the trap frequency along axis i, for $i = x$, y, z, and $a_{\mathrm{OH}} = \sqrt{\hbar/(m\overline{\omega})}$ is the harmonic oscillator length.

These scalings lead to the alternative dimensionless form of the GPE

$$\mathrm{i}\frac{\partial \psi}{\partial t} = -\frac{1}{2}\nabla^2\psi + V_{\mathrm{ext}}\psi + g|\psi|^2\psi - \mu\psi.$$

See appendix A for a full derivation. To solve this form of the GPE, set `eqn_to_solve` to `4` in `run.in`.

3.2.3 Alternative form

An alternative dimensionless form of the GPE can also be solved, specifically for the random phase approximation, as in Berloff & Svistunov (2002). This is

$$-2\mathrm{i}\frac{\partial\psi}{\partial t} = \nabla^2\psi + |\psi|^2\psi,$$

and it can be selected by setting `eqn_to_solve` to `2` in `run.in`.

3.3 Dimensionless forms of other quantities

It is useful to know the dimensionless forms of other relevant variables and quantities. These are described in this section. Full derivations of each of the non-dimensionalisations are given in appendix A.

3.3.1 Harmonic trapping potential

The dimensional harmonic trapping potential is given by

$$V_{\mathrm{ext}} = \frac{1}{2}m\left(\omega_x^2x^2 + \omega_y^2y^2 + \omega_z^2z^2\right). \tag{3.4}$$

Using the scaling $\omega_i \to \overline{\omega}\omega_i$, for $i = x,\ y,\ z$, and remembering to scale each coordinate, leads to the dimensionless form of the trapping potential

$$V_{\mathrm{ext}} = \frac{1}{2}\left(\omega_x^2x^2 + \omega_y^2y^2 + \omega_z^2z^2\right).$$

3.3.2 Thomas–Fermi approximation

The dimensional Thomas–Fermi approximation is given by

$$\psi = \sqrt{\frac{\mu - V_{\mathrm{ext}}}{g}}.$$

Since this approximation is only relevant for trapped condensates, we use harmonic oscillator units to non-dimensionalise. This leads to an identical dimensionless expression, where each dimensional variable is replaced by its dimensionless counterpart.

Condensate extent

The extent of the condensate R_i, $i = x,\ y,\ z$, in the Thomas–Fermi limit, is given by

$$R_i^2 = \frac{2\mu}{m\omega_i^2}, \quad i = x, y, z.$$

Non-dimensionalising using harmonic oscillator units yields

$$R_i^2 = \frac{2\mu}{\omega_i^2}, \quad i = x, y, z.$$

Number of atoms

The number of atoms within the condensate N, under the Thomas–Fermi approximation, is given by

$$N = \frac{8\pi}{15}\left(\frac{2\mu}{m\overline{\omega}^2}\right)^{\frac{3}{2}}\frac{\mu}{g}.$$

Again, non-dimensionalising using harmonic oscillator units, leads to

$$N = \frac{16\sqrt{2}\pi}{15}\frac{\mu^{\frac{5}{2}}}{g}.$$

3.3.3 Circulation

The dimensional circulation κ, around a vortex is defined to be

$$\kappa = \oint_C \boldsymbol{u} \cdot \mathrm{d}\boldsymbol{l}.$$

Then, using the fact that $\boldsymbol{u} = (\hbar/m)\boldsymbol{\nabla}\phi$, where ϕ is the phase, and also noting that the circulation is quantised, such that the phase differs by $2\pi n$ around the vortex, where n is the winding number, we obtain

$$\kappa = \frac{\hbar}{m}\oint_C \boldsymbol{\nabla}\phi \cdot \mathrm{d}\boldsymbol{l} = \frac{2\pi\hbar}{m}n.$$

Using natural units with the scaling $\kappa \to (2\mu a^2/\hbar)\kappa$, or harmonic oscillator units with the scaling $\kappa \to a_{\mathrm{OH}}^2\overline{\omega}\kappa$, in both cases leads to a dimensionless circulation of $\kappa = 2\pi n$.

3.4 Initial conditions

The code implements three basic initial conditions, any combination of which can theoretically be multiplied together. This section outlines the mathematical description of the initial conditions; for a description of the numerical implementation see §5.2.2.

3.4.1 Vortex line

A vortex line is given by

$$\psi_0 = f(r)\exp(\mathrm{i}\theta),$$

in polar coordinates (r, θ), where $f(r)$ is some function which models the vortex core, and θ is the phase. In the code, we use the vortex core model proposed by Berloff & Roberts (2001), so that

$$f(r) = 1 - \exp\left(-0.7r^{1.15}\right)$$

The vortex line initial condition can also support planar or helical perturbations along its length. For a vortex line oriented along the z-direction, this is achieved by imposing a sinusoidal perturbation of the form

$$\sin\left(\frac{2\pi(z - z_0)}{l}\right),$$

in the x-direction, and the same perturbation shifted by $\pi/2$ in the y-direction, where z_0 is the translation of the perturbation in the z-direction. These perturbations can be set independently; doing so will result in a planar perturbation, while setting both will result in a helical perturbation.

Cyclically permuting x, y, and z results in a vortex line along another direction, and the initial position and circulation of the line can also be defined. See §5.2.1 for a description of the parameters that control the vortex line initial condition.

3.4.2 Vortex ring

A vortex ring in the (y, z)-plane, travelling in the x-direction takes the form (Berloff, 2004)

$$\psi_0 = \Psi(x, s + R_0)\Psi^*(x, s - R_0),$$

where $\Psi(x, s)$ is given by

$$\Psi(x, s) = f\left(\sqrt{x^2 + s^2}\right)\exp(\mathrm{i}\theta),$$

$f(r)$ (which again models the vortex core) is given by

$$f(r)^2 = \frac{r^2\left(a_1 + a_2 r^2\right)}{1 + b_1 r^2 + b_2 r^4},$$

where a_i and b_i are constants, $s = \sqrt{y^2 + z^2}$, and R_0 is the radius of the ring.

The vortex ring initial condition also supports planar and helical perturbations, in a manner similar to that of the vortex line. The perturbed initial condition then takes the form

$$\begin{aligned}\psi_0 = \Psi\left\{x - A_1\cos(m_1\theta), s + R_0 - A_2\cos(m_2\theta)\right\} \times \\ \Psi^*\left\{x - A_1\cos(m_1\theta), s - R_0 - A_2\cos(m_2\theta)\right\},\end{aligned}$$

where $\Psi(x, s)$ is given by

$$\Psi(x, s) = f\left(\sqrt{x^2 + s^2}\right)(x + is)\,.$$

The function $f(r)$ is as above, $s = \sqrt{y^2 + z^2} - A_1 \sin(m_1\theta)$, A_1 and m_1 are the amplitude and wavenumber of a purely helical perturbation, and A_2 and m_2 are the amplitude and wavenumber of a purely planar perturbation.

Vortex rings can be set up in such a way that they are sitting in either of the three planes, and the direction of motion can also be defined. See §5.2.1 for a description of the parameters that control the vortex ring initial condition.

3.4.3 Random phase

The GPE can model the formation of a condensate, starting from a non-equilibrium initial condition. The random phase initial condition describes a weakly interacting Bose gas, where the particles remain in a strongly non-equilibrium state. Evolving this state leads to the formation of a quasi-condensate, consisting of a tangle of quantised vortices. This tangle decays as the system reaches thermal equilibrium, with a certain number of particles in the zero-momentum (genuine condensate) state. See, for example, Berloff & Svistunov (2002), Connaughton *et al.* (2005), and Berloff & Youd (2007) for more details on the random phase approximation.

To model this, the initial condition is set to

$$\psi_0 = \sum_{\boldsymbol{k}} a_{\boldsymbol{k}} \exp(\mathrm{i}\boldsymbol{k} \cdot \boldsymbol{r}),$$

where the $\boldsymbol{k}$s are the wavenumbers in momentum space, and the phases of the complex amplitudes $a_{\boldsymbol{k}}$ are distributed randomly.

The mass and kinetic energy density must be set in `run.in`, prior to performing a random phase simulation. These are `nv` and `enerv` respectively.

Chapter 4

Numerical formulation

This chapter outlines the numerical formulation which is used to solve the GPE. Time stepping is performed explicitly using a choice of schemes, and spatial discretisation is performed with either second- or fourth-order centred finite differences.

4.1 Time stepping

As briefly mentioned in the introduction, a choice of time stepping schemes is possible. Each of these is explained in this section.

4.1.1 Euler's method

For a general partial differential equation of the form

$$\frac{\partial \psi}{\partial t} = f(\psi, \boldsymbol{r}, t),$$

where f is some function representing the right hand side of the equation, Euler's method is

$$\psi^{p+1} = \psi^p + \Delta t f^p + \mathcal{O}(\Delta t^2),$$

where Δt is the time step, $\psi^p = \psi\left(x, y, z, t^p\right)$, $t^p = p\Delta t$, and $f^p = f\left(\psi^p\right)$.

This scheme is only first order accurate in Δt, and is not recommended for use, other than for very rough testing. It can be selected in the code by setting the parameter `scheme` equal to `euler` in `run.in`.

4.1.2 Second-order Runge–Kutta method

The second-order Runge–Kutta method (also known as the midpoint method) is second order accurate in time, and takes the form

$$\psi^{p+1} = \psi^p + \Delta t k_2 + \mathcal{O}(\Delta t^3),$$

where

$$k_1 = f\left(t^p, \psi^p\right),$$
$$k_2 = f\left(t^{p+\frac{1}{2}}, \psi^p + \frac{1}{2}\Delta t k_1\right).$$

This scheme can be selected by setting `scheme` equal to `rk2`.

4.1.3 Fourth-order Runge–Kutta method

The fourth-order Runge–Kutta method is fourth order accurate in time and takes the form

$$\psi^{p+1} = \psi^p + \Delta t\left(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}\right) + \mathcal{O}(\Delta t^5),$$

where

$$k_1 = f\left(t^p, \psi^p\right),$$
$$k_2 = f\left(t^{p+\frac{1}{2}}, \psi^p + \frac{1}{2}\Delta t k_1\right),$$
$$k_3 = f\left(t^{p+\frac{1}{2}}, \psi^p + \frac{1}{2}\Delta t k_2\right),$$
$$k_4 = f\left(t^{p+1}, \psi^p + \Delta t k_3\right).$$

This scheme can be selected by setting `scheme` equal to `rk4`.

4.1.4 Runge–Kutta–Fehlberg method

The Runge–Kutta–Fehlberg method is a hybrid fourth/fifth order adaptive time stepping scheme. In the code, the fifth-order formula is

$$\begin{aligned}
k_1 &= \Delta t f\left(t^p, \psi^p\right),\\
k_2 &= \Delta t f\left(t^p + a_2\Delta t, \psi^p + b_{21}k_1\right),\\
k_3 &= \Delta t f\left(t^p + a_3\Delta t, \psi^p + b_{31}k_1 + b_{32}k_2\right),\\
k_4 &= \Delta t f\left(t^p + a_4\Delta t, \psi^p + b_{41}k_1 + b_{42}k_2 + b_{43}k_3\right),\\
k_5 &= \Delta t f\left(t^p + a_5\Delta t, \psi^p + b_{51}k_1 + b_{52}k_2 + b_{53}k_3 + b_{54}k_4\right),\\
k_6 &= \Delta t f\left(t^p + a_6\Delta t, \psi^p + b_{61}k_1 + b_{62}k_2 + b_{63}k_3 + b_{64}k_4 + b_{65}k_5\right),\\
\psi^{p+1} &= \psi^p + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 + \mathcal{O}(\Delta t^6).
\end{aligned}$$

The embedded fourth-order formula is given by

$$\psi_*^{p+1} = \psi^p + c_1^*k_1 + c_2^*k_2 + c_3^*k_3 + c_4^*k_4 + c_5^*k_5 + c_6^*k_6 + \mathcal{O}(\Delta t^5),$$

so that an error estimate can be obtained with

$$\Delta \equiv \psi^{p+1} - \psi_*^{p+1} = \sum_{i=1}^{6}\left(c_i - c_i^*\right)k_i.$$

i	a_i	b_{ij}					c_i	c_i^*
1							$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$	$\frac{13525}{55296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$
$j=$		1	2	3	4	5		

Table 4.1: The Cash–Karp constants for the Runge–Kutta–Fehlberg scheme.

The values of the constants are those given by Cash & Karp (1990), and are reproduced here in table 4.1. Full details of the algorithm can be found in Numerical Recipes (§16.2, p.708, Press *et al.*, 1992).

This scheme can be selected by setting `scheme` equal to `rk45`.

4.2 Spatial discretisation

Centred finite differences are used to spatially discretise the GPE; these can be either second or fourth order.

For a general partial differential equation of the form

$$\frac{\partial \psi}{\partial t} = f(\psi, \boldsymbol{r}, t),$$

where f is some function representing the right hand side of the equation, the spatial discretisation takes the form

$$\frac{\partial \psi_{ijk}}{\partial t} = f_{ijk},$$

where $\psi_{ijk} = \psi\left(x_i, y_j, z_k, t\right)$, $x_i = i\Delta x$, $y_j = j\Delta y$, $z_k = k\Delta z$, and $f_{ijk} = f\left(\psi_{ijk}\right)$.

4.2.1 Second-order finite differences

The second-order finite-difference approximation to the first derivative, with respect to x, is given by

$$\frac{\partial \psi}{\partial x} \approx \frac{\psi_{i+1} - \psi_{i-1}}{2\Delta x},$$

where we have dropped the j and k indices for clarity.

Similarly, the second-order approximation to the second derivative is

$$\frac{\partial^2 \psi}{\partial x^2} \approx \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{\Delta x^2}.$$

Second-order finite differences can be chosen by setting `order` equal to `2` in `run.in`.

4.2.2 Fourth-order finite differences

The fourth-order finite-difference approximation to the first derivative, with respect to x, is given by

$$\frac{\partial \psi}{\partial x} \approx \frac{-\psi_{i+2} + 8\psi_{i+1} - 8\psi_{i-1} + \psi_{i-2}}{12\Delta x}.$$

Similarly, the fourth-order approximation to the second derivative is

$$\frac{\partial^2 \psi}{\partial x^2} \approx \frac{-\psi_{i+2} + 16\psi_{i+1} - 30\psi_i + 16\psi_{i-1} - \psi_{i-2}}{12\Delta x^2}.$$

Fourth-order finite differences can be chosen by setting `order` equal to `4`.

4.3 Boundary conditions

The code implements both periodic and reflective boundary conditions. Since both of these conditions are behavioural, rather than numerical, no boundary conditions are explicitly set on ψ itself. Note that it is not possible to mix boundary conditions yet.

4.3.1 Periodic boundary conditions

Suppose the x-coordinate runs from x_0 to x_n. Then clearly the values of ψ at x_{-1} and x_{n+1} are needed to compute a first derivative, for example.

Periodic boundary conditions are implemented such that

$$x_{-1} = x_n,$$
$$x_{n+1} = x_0.$$

These conditions are also applied in the y- and z-directions, and can be naturally extended for higher order approximations.

To use periodic boundary conditions set `bcs` equal to `1` in `run.in`.

4.3.2 Reflective boundary conditions

The physical meaning of reflective boundary conditions is that structures within the computational box see images of themselves across the box boundaries. This would cause a vortex ring to annihilate on approaching a boundary, for example, as it sees an image of itself approaching the boundary from the opposite direction (outside the computational box).

These boundary conditions are implemented such that

$$x_{-1} = x_1,$$
$$x_{n+1} = x_{n-1}.$$

To use reflective boundary conditions set `bcs` equal to `2`.

Chapter 5

File reference

This chapter is intended as a reference for the main files associated with the code, such as the source files, input and output files, and the IDL `gpe.pro` program.

5.1 Program source files

Table 5.1 lists the `.f90` source code files which make up the code, and describes their functionality.

File	Description
`constants.f90`	This is a module which defines the constants needed for FFTw.
`derivs.f90`	Routines to do with derivatives.
`error.f90`	Error-handling routines.
`gpe.f90`	Main program file.
`ic.f90`	Routines to do with setting up the initial condition and general initialisation.
`parameters.f90`	Compile-time parameters and global variables.
`solve.f90`	Routines to do with actually solving the equation, for example, the time stepping algorithms are defined in this file.
`variables.f90`	User-defined types, and other general routines to do with the equation variables.

Table 5.1: Program source files which make up the GPE code.

5.2 Program input files

This section describes the `.in` input files which will generally need to be edited to set up a run.

5.2.1 parameters.in

Edit this file to set the floating-point precision, the number of processes, and the grid dimensions of the run. If the initial condition consists of a vortex line or vortex ring, then the line/ring parameters can also be set here. Note that any changes to this file will require a recompilation of the code. See tables 5.2, 5.3, and 5.4 for a description of the parameters.

Parameter	Type	Description
`pr`	integer	The precision of real variables is parametrised. Choose the desired line for either real or double precision.
`nyprocs`	integer	The number of processes in the y-direction.
`nzprocs`	integer	The number of processes in the z-direction.
`nx`	integer	The number of grid points in the x-direction.
`ny`	integer	The number of grid points in the y-direction.
`nz`	integer	The number of grid points in the z-direction.

Table 5.2: Compile-time parameters to set.

There are no restrictions on the `nyprocs` and `nzprocs` parameters, other than that they be $\geqslant 1$. In general, it is recommended that `nzprocs` $\geqslant$ `nyprocs` for best performance, so that fewer non-contiguous data transfers are performed. They can both be set to 1, in which case the job is simply run on one process (an MPI parallel environment is still required though).

New vortex lines and rings can be defined simply by copying an existing definition in `parameters.in`, and renaming, so for example, the *ring* example defines

```
type (ring_param), parameter :: &
  vr1 = ring_param('yz', 0.0_pr, 0.0_pr, 0.0_pr, 10.0_pr, &
    0.0_pr, 5, 0.0_pr, 10, -1.0_pr)
```

To create another vortex ring definition, say with a radius of 20, situated at $x = 5$, and sitting in the (x, y)-plane, define

```
type (ring_param), parameter :: &
  vr2 = ring_param('xy', 5.0_pr, 0.0_pr, 0.0_pr, 20.0_pr, &
    0.0_pr, 5, 0.0_pr, 10, -1.0_pr)
```

Then an initial condition consisting of these two vortex rings could be set up as described in the next section.

5.2.2 ic.in

This file defines the initial condition. The initial condition must be defined in the form `init_cond = function()`, where `function()` is some function in the code which defines a possible component of an initial condition. Components can be

Parameter	Type	Description
x0	real	x-position of the line (or translation in the x-direction if dir='x').
y0	real	y-position of the line (or translation in the y-direction if dir='y').
z0	real	z-position of the line (or translation in the z-direction if dir='z').
amp1	real	Amplitude of a sinusoidal disturbance in one direction along the line.
amp2	real	Amplitude of a sinusoidal disturbance in the other direction along the line.
ll	real	The wavelength of the above disturbances.
sgn	real	The sign of the argument of the line (i.e. circulation direction).
dir	character	The direction (x, y, or z) in which the line should extend.
imprint_phase	logical	Whether only the phase should be imprinted, i.e. no vortex core should be modelled.

Table 5.3: Compile-time parameters for a vortex line.

multiplied together to form any number of different initial conditions. Look at ic.in in the src directory to see some examples.

In the *ring* example, the initial condition is set to vortex_ring(vr1), where vr1 is a type parameter declared in parameters.in (see above).

As another example, if you define vr2 as above, then you could construct an initial condition consisting of two rings with

```
init_cond = vortex_ring(vr1) * vortex_ring(vr2)
```

In this way, any number of initial conditions can be constructed, simply by multiplying functions together.

As with parameters.in, any changes to this file will require the code to be recompiled.

5.2.3 run.in

This file defines the main parameters for the run, as a set of Fortran namelists. Each namelist loosely collects together related parameters. The namelists are:

- run_params — these are parameters to do with the run itself, such as time step, time stepping scheme, when the run should end, which equation to solve, etc.;

Parameter	Type	Description
`plane`	character	Plane in which the ring should sit (`'xy'`, `'xz'`, or `'yz'`).
`x0`	real	x-position of the ring.
`y0`	real	y-position of the ring.
`z0`	real	z-position of the ring.
`r0`	real	Radius of the ring.
`amp`	real	Amplitude of a planar disturbance around the ring.
`mm`	integer	Wavenumber of a planar disturbance.
`r1`	real	Amplitude of a helical disturbance around the ring.
`kk`	integer	Wavenumber of a helical disturbance.
`dir`	real	Ring propagation direction (± 1).

Table 5.4: Compile-time parameters for a vortex ring.

- `eqn_params` — these parameters set properties of the equation, such as whether it should be solved in a moving reference frame, trap parameters, random phase parameters, etc.;
- `io_params` — these parameters control input/output, for example, what data should be saved and how often;
- `misc_params` — miscellaneous parameters which do not fit in the other categories.

Table 5.5 describes these parameters in detail.

Parameter	Type	Description
`tau`	real	Time step (initial time step for RK45). Valid for both imaginary and real time.
`end_time`	real	The final (dimensionless) time.
`xr`	real	The x-coordinate of the right-hand-side of the computational box (the left-hand-side is set to `-xr`).
`yr`	real	As above but for the y-coordinate.
`zr`	real	As above but for the z-coordinate.
`scheme`	character	The time stepping scheme to use. This must be set to one of `euler` (for explicit second-order Euler time stepping), `rk2` (for explicit second-order Runge–Kutta time stepping), `rk4` (for explicit fourth-order Runge–Kutta time stepping), or `rk45` (for explicit adaptive fourth-order Runge–Kutta–Fehlberg time stepping).
`eqn_to_solve`	integer	The form of the GPE to solve. See §3.2 for the possible values to use.

`bcs`	integer	The boundary conditions to use. Set to `1` for periodic BCs; set to `2` for reflective BCs.
`order`	integer	The order of the derivatives to use. Set to `2` for second-order; set to `4` for fourth-order.
`restart`	logical	Set to `.true.` to do a restart of a previous run. See §2.9.
`saved_restart`	logical	Set to `.true.` if using filtered data from a previous run to multiply with the initial condition of a new run.
`multiply_ic_restart`	logical	Set to `.true.` if a restarted run should be multiplied by the initial condition defined in `ic.in`. See §2.9.1.
`renorm`	logical	Set to `.true.` if the wavefunction should be renormalised at every time step in imaginary time.
`imprint_vl`	logical	Set to `.true.` if the wavefunction should be multiplied by a vortex line at each time step in imaginary time.
`stop_imag`	logical	Set to `.true.` if the run should stop at the end of imaginary time, i.e. when the relative norms of successive time steps are deemed to be sufficiently close (currently 10^{-12}).
`real_time`	logical	Set to `.true.` if the run should be started in real time.
`Urhs`	real	Set non-zero to solve the equation in a moving reference frame.
`diss_amp`	real	Set non-zero to include dissipation of this amplitude at the boundaries.
`scal`	real	Set non-zero to scale vortex rings/lines.
`nv`	real	For random phase approximation, the total mass per unit volume.
`enerv`	real	For random phase approximation, the total kinetic energy per unit volume.
`g`	real	Interaction parameter for trapped condensate.
`mu`	real	Chemical potential for trapped condensate.
`nn`	real	Number of atoms in a trapped condensate. Should currently be left to `1.0`, and instead tune `mu` and `g` to specify `nn`.
`omx`	real	Frequency of trap in x-direction.
`omy`	real	Frequency of trap in y-direction.
`omz`	real	Frequency of trap in z-direction.
`save_rate`	integer	The rate at which time-series data should be saved (roughly corresponding to the number of time steps).

save_rate2	real	How often (in terms of actual time units) isosurface data should be saved (3D isosurfaces, 2D surfaces).
save_rate3	real	How often (in terms of actual time units) data to do with condensed particles and PDFs should be saved.
p_save	real	How often (in terms of actual time units) the code should save its own state, so that it can be restarted in the event of a machine failure, for example.
save_contour	logical	Should 2D contour data be saved?
save_3d	logical	Should 3D isosurface data be saved?
save_filter	logical	Should 3D filtered isosurfaces of the density be saved?
filter_kc	real	The cutoff wavenumber used to filter the isosurfaces.
save_average	logical	Should 3D time-averaged isosurfaces of the density be saved?
save_spectrum	logical	Should various spectra be saved (mainly for random phase approximation)?
save_pdf	logical	Should PDFs of the velocity components be saved?
save_vcf	logical	Should the velocity correlation function be saved?
save_ll	logical	Should the vortex line length be saved?
save_zeros	logical	Should the points of zero density be saved? (Not reliable!)

Table 5.5: Run-time parameters to set.

5.3 Makefile

Settings in the Makefile may need to be changed, depending on the architecture on which the code is run, and what compilers are available to you. Table 5.6 describes the Makefile variables.

5.4 Program output files

The output that the program produces depends on the parameters set in the io_params namelist. Table 5.7 briefly describes these files.

Variable	Description
`OBJECT`	The name of the executable produced on compilation.
`OBJS`	The object files that should be linked.
`FC`	The Fortran compiler to be used. This could be the MPI wrapper compiler `mpif90`, but the underlying Fortran compiler must be able to compile the code (e.g. sunf95, ifort, gfortran).
`FFLAGS`	Compiler flags. See the compiler's manual. Sunf95 works well with `-fast`. If nonsense results are produced `-fsimple=0` might be required.
`LDFFTW`	FFTw libraries to link. Compiling with `make precision=single` will link the single precision libraries.
`LDFLAGS`	Any extra flags required by the linker. If `mpif90` is not the compiler, then all the MPI libraries will need to be linked.
`INCLUDE`	Include path (e.g. for MPI header files).

Table 5.6: Description of Makefile variables.

5.4.1 Binary layout

If you do not have access to IDL, but still want to be able to view contour and isosurface plots, then you will need to know how this data is arranged in the numbered `dens*******.dat` files, within each `proc` directory. Table 5.8 describes the data which are saved to these files, and how much space (in terms of bytes) is needed to store the data.

5.5 run.sh

This script can be used to start a run on a shared memory machine, such as those with the latest multi-core processors, or older multi-processor machines. Usage instructions are provided with the script itself; run with no arguments to see the help.

5.6 The IDL gpe.pro program

If you have access to IDL, then all of the contour and isosurface visualisation can be done through the `gpe.pro` program. This section will briefly describe its use. Detailed examples of some aspects of the program are given in §2.8.

5.6.1 Floating point precision

It is important to note the precision of the run for which you want to view results. If you have performed a double-precision run, then all `gpe` commands must include the `/dbl` keyword.

5.6.2 Fortran unformatted I/O

The GPE code used to write binary data using Fortran 77 unformatted, record-based I/O. With the advent of Fortran 2003 stream I/O, this is no longer necessary, and record markers are not part of the data. It might still be necessary to view the old data, however, in which case the `/f77` keyword should be used in all `gpe` commands.

5.6.3 Isosurface plots

The program will produce an isosurface plot of the density $|\psi|^2$ if no keywords or options are provided, e.g.

```
gpe, 0, 0
```

To change the default surface colour, set the `index` option, e.g.

```
gpe, 0, 0, index=100
```

The `index` option must be in the range 0 to 255, and corresponds to the index into the currently loaded IDL colour table. The isosurface level can also be controlled, by using the `level` option, e.g.

```
gpe, 0, 0, level=0.4
```

5.6.4 Contour plots

A contour plot of the density in the (x, y)-plane at $z = 0$ is displayed with the addition of the `/cntr` keyword, e.g.

```
gpe, 0, 0, /cntr
```

Plots of the phase or velocities can also be produced by adding the `/phase`, `/vx`, `/vy`, or `/vz` keywords, e.g.

```
gpe, 0, 0, /cntr, /phase
```

The position of the contour slice can be controlled by using the `xpos`, `ypos`, or `zpos` options, e.g.

```
gpe, 0, 0, /cntr, xpos=2.0
```

The position is given in real units (as opposed to grid units). The plane in which the contour slice sits can be controlled with the `dir` option, e.g.

```
gpe, 0, 0, /cntr, dir='y'
```

This will produce a contour plot in the (x, z)-plane.

5.6.5 Slice plots

One-dimensional slices through the data can also be generated, by using the `/slice` keyword, and the position and direction controlled as for contour plots, e.g.

```
gpe, 0, 0, /slice, xpos=3.4, dir='z'
```

5.6.6 Contour animations

A series of contour snapshots can be generated, by using the `/c_anim` keyword, e.g.

```
gpe, 0, 9, /cntr, /phase, /c_anim
```

Snapshots are saved to the directory `images`, from the directory under which IDL is started, so this directory must exist before attempting to create snapshots, otherwise an error will result.

5.6.7 Isosurface animations

A series of isosurface snapshots can be generated, by using the `/png` keyword, e.g.

```
gpe, 0, 9, /png
```

Currently, it is only possible to generate isosurfaces of the density.

5.6.8 EPS output

High quality EPS figures of all 1D, 2D, and 3D plots can be produced, by using the `/eps` keyword. Figures are saved in the images directory (as for the snapshots in the animations above), so this directory must exist prior to attempting to save as EPS.

5.6.9 Saving VAPOR data

The `gpe.pro` program can save data in a form suitable for post-processing by VAPOR — the visualisation and analysis platform, often used by ocean, atmosphere, and solar researchers (`http://www.vapor.ucar.edu/`).

The bulk of the work to do this is performed by the IDL `save_vapor_data.pro` program in the `idl/utils` directory. This program calls auxiliary IDL routines which are only available once VAPOR is installed, therefore, you must have a functioning VAPOR installation, prior to attempting to save VAPOR data. See the VAPOR website for installation and setup instructions.

Once VAPOR is installed, suitable data can be saved with the `/vapor` keyword, e.g.

```
gpe, 0, 9, /vapor
```

This will create a `gpe.vdf` VDF metafile (which describes the data), in a sub-directory `vapor` (which must already exist). Also created within the `vapor` directory, is a directory `gpe_data`, which includes further sub-directories where the VAPOR data resides. (The number of directories here depends on which data you requested to be saved.)

The density data are always saved by default. In addition, you can request that the phase, or the velocities are also saved, using the relevant keywords (as explained above).

By default, two refinement levels of the data are saved. This can be altered with the `num_levels` keyword, e.g.

```
gpe, 0, 9, /vapor, num_levels=1
```

which will save only one refinement level.

If you subsequently decide to save more data for VAPOR post-processing, for example, if you have continued a run, and don't want to save all the data again, then you can add the `/append` keyword, to add the extra information to the VDF metafile. So, continuing from the previous example,

```
gpe, 10, 19, /vapor, /append
```

would add files 10–19 to the VAPOR data. To actually view the data, load the `gpe.vdf` file into VAPOR.

Important note: VAPOR does not yet support double precision floating point arithmetic, so even if your run is performed in double precision, the data you view with VAPOR is only in single precision. You must still provide the `/dbl` keyword as normal, but all double precision variables are silently converted to single precision.

File	Description
`energy.dat`	Saves the energy at each time.
`linelength.dat`	Saves the total line length of vortices in the condensate.
`mass.dat`	Saves the mass at each time.
`minmax_*.dat`	Saves the minimum and maximum of the density, filtered density, and time-averaged density, over the duration of the run.
`norm.dat`	Saves the norm at each time.
`misc.dat`	Any miscellaneous data can be sent to this file.
`momentum.dat`	Saves the three components of the momentum.
`p_saved.dat`	The values of the time index `p` when the code saved its own state.
`save.dat`	The parameters for the most recently saved state.
`timestep.dat`	The imaginary and real time step at each time, if adaptive time stepping is chosen.
`psi_time.dat`	Saves the real and imaginary time, the real and imaginary parts of the wavefunction, the density, and the phase.
`proc**`	Numbered directories corresponding to each process involved in the run. Each of these directories contains the binary files listed below.
`end_state.dat`	The saved state of the run.
`im_zeros*******.dat`	The coordinates where the imaginary part of the wavefunction goes to zero.
`re_zeros*******.dat`	The coordinates where the real part of the wavefunction goes to zero.
`dens*******.dat`	If 3D isosurfaces are requested, the data for them are saved in these files.
`filtered*******.dat`	As above, but for filtered data.
`ave*******.dat`	As above, but for time-averaged data.
`spectrum*******.dat`	The spectrum ($n_{\boldsymbol{k}}$ vs. $\boldsymbol{k}$).
`zeros*******.dat`	The coordinates where the real and imaginary parts of the wavefunction simultaneously go to zero.

Table 5.7: Output files from the GPE code.

Variable	Type	Size (bytes)		Description
		Single	Double	
`t+im_t`	real	4	8	The total time elapsed.
`nx`	integer	4	4	Number of grid points in the x-direction.
`ny`	integer	4	4	Number of grid points in the y-direction.
`nz`	integer	4	4	Number of grid points in the z-direction.
`nyprocs`	integer	4	4	Number of processes in the y-direction.
`nzprocs`	integer	4	4	Number of processes in the z-direction.
`js`	integer	4	4	Starting index of data in y-direction, local to each process.
`je`	integer	4	4	Ending index of data in y-direction, local to each process.
`ks`	integer	4	4	Starting index of data in z-direction, local to each process.
`ke`	integer	4	4	Ending index of data in z-direction, local to each process.
`psi`	complex	8`nx*nyl*nzl`	16`nx*nyl*nzl`	Complex wavefunction ψ, local to each process.
`x`	real	4`nx`	8`nx`	The grid array in the x-direction.
`y`	real	4`ny`	8`ny`	The grid array in the y-direction.
`z`	real	4`nz`	8`nz`	The grid array in the z-direction.

Table 5.8: Data saved to the `dens*******.dat` files, with sizes in bytes for both single and double precision (assuming x86 or x86-64 architectures). The number of grid points local to each process in the y- and z-directions is given by `nyl=je-js` and `nzl=ke-ks`.

Appendix A

Derivations of non-dimensionalised equations and variables

This chapter derives the non-dimensionalisations of the GPE, and related quantities, in more detail.

A.1 Physical units

The following table lists the physical units for various quantities related to the GPE and Bose–Einstein condensation.

Quantity	Description	Unit
a	Healing length	m
a_{OH}	Harmonic oscillator length	m
$\hbar$	Reduced Planck constant	Js
m	Mass	$\mathrm{Js^2m^{-2}}$
μ	Chemical potential	J
g	Interaction parameter	$\mathrm{Jm^3}$
ψ	Complex wavefunction	$\mathrm{m}^{-\frac{3}{2}}$
ω	Trap frequency	$\mathrm{s^{-1}}$
V_{ext}	Trapping potential	J
κ	Circulation	$\mathrm{m^2s^{-1}}$

Table A.1: Physical units for quantities related to the GPE and Bose–Einstein condensation.

A.2 Natural units non-dimensionalisation

Using the scalings given by (3.2), the dimensional GPE (3.1) becomes (ignoring the trapping potential V_{ext})

$$\mathrm{i}\hbar\frac{2\mu}{\hbar}\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{1}{a^2}\nabla^2\psi + g\psi_\infty^2|\psi|^2\psi - \mu\psi.$$

Each term in the GPE involves a term in ψ. The factor of ψ_∞ used to non-dimensionalise ψ is therefore cancelled throughout.

Cancelling $\hbar$ on the left, and dividing through by μ leads to

$$2\mathrm{i}\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2\mu m}\frac{1}{a^2}\nabla^2\psi + \frac{g}{\mu}\psi_\infty^2|\psi|^2\psi - \psi.$$

Then

$$-\frac{\hbar^2}{2\mu m}\frac{1}{a^2} = -\frac{\hbar^2}{2\mu m}\frac{2\mu m}{\hbar^2} = -1,$$

and

$$\frac{g}{\mu}\psi_\infty^2 = \frac{g}{\mu}\frac{\mu}{g} = -1,$$

and so the non-dimensionalised GPE using natural units is

$$-2\mathrm{i}\frac{\partial\psi}{\partial t} = \nabla^2\psi + \left(1 - |\psi|^2\right)\psi.$$

A.3 Harmonic oscillator units non-dimensionalisation

Using the scalings given by (3.3), the dimensional GPE (3.1) becomes

$$\mathrm{i}\hbar\overline{\omega}\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{1}{a_{\text{OH}}^2}\nabla^2\psi + \hbar\overline{\omega}V_{\text{ext}}\psi + a_{\text{OH}}^3\hbar\overline{\omega}g\frac{1}{a_{\text{OH}}^3}|\psi|^2\psi - \hbar\overline{\omega}\mu\psi,$$

where again, the scaling factor for ψ is cancelled throughout.

Then

$$-\frac{\hbar^2}{2m}\frac{1}{a_{\text{OH}}^2} = -\frac{\hbar^2}{2m}\frac{m\overline{\omega}}{\hbar} = -\frac{1}{2}\hbar\overline{\omega},$$

and

$$a_{\text{OH}}^3\hbar\overline{\omega}g\frac{1}{a_{\text{OH}}^3} = \hbar\overline{\omega}g.$$

Then, dividing through by $\hbar\overline{\omega}$ leads to the GPE, non-dimensionalised using harmonic oscillator units

$$\mathrm{i}\frac{\partial\psi}{\partial t} = -\frac{1}{2}\nabla^2\psi + V_{\text{ext}}\psi + g|\psi|^2\psi - \mu\psi.$$

A.4 Harmonic trapping potential

Using the scalings $V_{\text{ext}} \to \hbar\overline{\omega}V_{\text{ext}}$, $\omega_i \to \overline{\omega}\omega_i$, for $i = x,\ y,\ z$, and $\boldsymbol{r} \to a_{\text{OH}}\boldsymbol{r}$, the dimensional trapping potential (3.4) becomes

$$\begin{aligned}\hbar\overline{\omega}V_{\text{ext}} &= \frac{1}{2}ma_{\text{OH}}^2\overline{\omega}^2\left(\omega_x^2x^2 + \omega_y^2y^2 + \omega_z^2z^2\right)\\ &= \frac{1}{2}m\frac{\hbar}{m\overline{\omega}}\overline{\omega}^2\left(\cdots\right)\\ &= \frac{1}{2}\hbar\overline{\omega}\left(\cdots\right),\end{aligned}$$

and therefore, on dividing through by $\hbar\overline{\omega}$, the dimensionless trapping potential is

$$V_{\text{ext}} = \frac{1}{2}\left(\omega_x^2x^2 + \omega_y^2y^2 + \omega_z^2z^2\right).$$

A.5 Thomas–Fermi approximation

Starting from

$$|\psi|^2 = \frac{\mu - V_{\text{ext}}}{g},$$

and scaling with harmonic oscillator units, leads to

$$\frac{1}{a_{\text{OH}}^3}|\psi|^2 = \frac{\hbar\overline{\omega}\mu - \hbar\overline{\omega}V_{\text{ext}}}{a_{\text{OH}}^3\hbar\overline{\omega}g},$$

so that the dimensionless form of the Thomas–Fermi approximation is

$$|\psi|^2 = \frac{\mu - V_{\text{ext}}}{g}.$$

A.5.1 Condensate extent

Starting from

$$R_i^2 = \frac{2\mu}{m\omega_i^2},$$

for $i = x,\ y,\ z$, and again scaling using harmonic oscillator units, gives

$$\begin{aligned}a_{\text{OH}}^2R_i^2 &= \frac{2\hbar\overline{\omega}\mu}{m\overline{\omega}^2\omega_i^2}\\ &= \frac{2a_{\text{OH}}^2\overline{\omega}\mu}{\overline{\omega}\omega_i^2},\end{aligned}$$

and, on dividing through by a_{OH}^2, we have the dimensionless condensate extent

$$R_i^2 = \frac{2\mu}{\omega_i^2}.$$

A.5.2 Number of atoms

The number of atoms within the condensate, under the Thomas–Fermi approximation, is given by

$$N = \frac{8\pi}{15}\left(\frac{2\mu}{m\overline{\omega}^2}\right)^{\frac{3}{2}}\frac{\mu}{g}.$$

Non-dimensionalising gives

$$\begin{aligned}
N &= \frac{8\pi}{15}\left(\frac{2\hbar\overline{\omega}\mu}{m\overline{\omega}^2}\right)^{\frac{3}{2}}\frac{\hbar\overline{\omega}\mu}{a_{\mathrm{OH}}^3\hbar\overline{\omega}g} \\
&= \frac{8\pi}{15}\left(2a_{\mathrm{OH}}^2\mu\right)^{\frac{3}{2}}\frac{\mu}{a_{\mathrm{OH}}^3 g} \\
&= \frac{8\pi}{15}\left(2\mu\right)^{\frac{3}{2}}\frac{\mu}{g} \\
&= \frac{8\pi}{15}2^{\frac{3}{2}}\frac{\mu^{\frac{5}{2}}}{g} \\
&= \frac{16\sqrt{2}\pi}{15}\frac{\mu^{\frac{5}{2}}}{g}.
\end{aligned}$$

A.6 Circulation

The dimensional circulation is given by

$$\kappa = \frac{h}{m} = \frac{2\pi\hbar}{m}.$$

Letting $\kappa \to (2\mu a^2/\hbar)\kappa$, and scaling with natural units, gives

$$\begin{aligned}
\frac{2\mu a^2}{\hbar}\kappa &= \frac{2\pi\hbar}{m} \\
\implies \kappa &= \frac{2\pi\hbar^2}{2\mu a^2 m} \\
&= \frac{2\pi a^2}{a^2} \\
&= 2\pi.
\end{aligned}$$

Simlarly, letting $\kappa \to a_{\mathrm{OH}}^2\overline{\omega}\kappa$, and scaling with harmonic oscillator units, gives

$$\begin{aligned}
a_{\mathrm{OH}}^2\overline{\omega}\kappa &= \frac{2\pi\hbar}{m} \\
\implies \kappa &= \frac{2\pi\hbar}{a_{\mathrm{OH}}^2\overline{\omega}m} \\
&= \frac{2\pi a_{\mathrm{OH}}^2}{a_{\mathrm{OH}}^2} \\
&= 2\pi.
\end{aligned}$$

Bibliography

Al-Amri, S. Z. Z., Youd, A. J. & Barenghi, C. F. 2008 Reconnection of superfluid vortex bundles. *Phys. Rev. Lett.* **101**, 215302.

Berloff, N. G. 2004 Padé approximations of solitary wave solutions of the Gross–Pitaevskii equation. *J. Phys. A: Math. Gen.* **37**, 1617–1632.

Berloff, N. G. & Roberts, P. H. 2001 Motion in a Bose condensate: IX. Crow instability of antiparallel vortex pairs. *J. Phys. A: Math. Gen.* **34**, 10057–10066.

Berloff, N. G. & Svistunov, B. V. 2002 Scenario of stronly nonequilibrated Bose–Einstein condensation. *Phys. Rev. A* **66**, 013603.

Berloff, N. G. & Youd, A. J. 2007 Dissipative dynamics of superfluid vortices at non-zero temperatures. *Phys. Rev. Lett.* **99**, 145301.

Cash, J. R. & Karp, A. H. 1990 A variable order Runge–Kutta method for initial value problems with rapidly varying right-hand sides. *ACM Transactions on Mathematical Software* **16**, 201–222.

Connaughton, C., Josserand, C., Picozzi, A., Pomeau, Y. & Rica, S. 2005 Condensation of classical nonlinear waves. *Phys. Rev. Lett.* **95**, 263901.

Gross, E. P. 1961 Structure of a quantized vortex in boson systems. *Il Nuovo Cimento* **20**, 454–457.

Helm, J. L., Barenghi, C. F. & Youd, A. J. 2011 Slowing down of vortex rings in Bose–Einstein condensates. *Phys. Rev. A* **83**, 045601.

Pitaevskii, L. P. 1961 Vortex lines in an imperfect Bose gas. *J. Exptl. Theoret. Phys. (U. S. S. R.)* **13**, 451–454.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. 1992 *Numerical recipes in Fortran 77*, 2nd edn. Cambridge University Press.

Tebbs, R., Youd, A. J. & Barenghi, C. F. 2011 The approach to vortex reconnection. *J. Low Temp. Phys.* **162**, 314–321.

White, A. C., Barenghi, C. F., Proukakis, N. P., Youd, A. J. & Wacks, D. H. 2010 Non-classical velocity statistics in a turbulent atomic Bose–Einstein condensate. *Phys. Rev. Lett.* **104**, 075301.