GANDALF v0.4.0 userguide

David Hubber, Giovanni Rosotti & Richard Booth

September 25, 2018

Contents

1 Overview of code

GANDALF is a new SPH and N-body code written in C++ and Python. Although partly based on some of the algorithms and code structures used in SEREN (Hubber et al. 2011), it has been written from scratch and contains many new features and optimisations which significantly improve the speed, functionality and usability of the code. It has been written for several purposes.

- GANDALF has been written with a fully object-oriented philosophy in C++. This is to improve the maintainability of the code and also to allow the code to be easily extended in the future with as little disruption to the original code-base as possible.
- GANDALF can be run in three different modes; as a standard C++ executable run from the command line, run from a python script or run inside an interactive python environment. Both the script and interactive python modes include a graphical output that can be used to visualise previously run simulations, or to interactively visualise simulations as they are run on-the-fly.
- GANDALF contains a variety of SPH algorithms such as conservative 'grad-h' SPH (Springel & Hernquist 2002, Price & Monaghan 2004) and Saitoh & Makino (2013).
- The N-body component of GANDALF contains a variety of common integration schemes such as 4th/6th-order Hermite with time-symmetric integration.
- GANDALF can generate initial conditions at run-time, as opposed to SEREN where the initial conditions had to be prepared a priori to running the simulation. The initial conditions may either be hard-coded into the C++ code, or set-up via a python script using numpy arrays.
- GANDALF uses additional parameters to switch on various physics options, as well as requiring them to be activated in the Makefile. In comparison, SEREN controlled physics options exclusively in the Makefile. Therefore, GANDALF need not be re-compiled everytime a different setting is required.

GANDALF has been developed as a fully open-source project hosted on the site github at the address `https://github.com/gandalfcode/gandalf`. Since GANDALF is currently in the beta-testing phase, it would be very helpful for the authors if new users could provide feedback of any problems with the code, or with any suggestions for improvements of existing features. Ideally, users would submit bug reports to the GANDALF github page (`https://github.com/gandalfcode/gandalf/issues`) since other users can see that a bug has been identified before sending duplicate bug reports.

2 Download & Installation

The easiest way to download the code is using git. Just cd to the folder you prefer and type:

```
git clone https://github.com/gandalfcode/gandalf.git
```

A new folder called "gandalf" will be created in the current directory and a fresh version of the code will be downloaded inside it. The use of the software version control system git makes easy to stay up-to-date with the changes in the code while keeping your own version. If you want to know more about git, it is plenty of tutorials on-line. A good starting point is `http://git-scm.com/book/en/v2`.

2.1 Requirements

If you are not interested in the Python library, you need only a C++ compiler. In order to use all features in GANDALF, the following programs and libraries must be installed :

- C++ compiler
- Python 2.7
- swig compatible with python 2.7
- matplotlib compatible with python 2.7
- numpy
- scipy

To reduce the number of needed libraries, we ship together with GANDALF the following packages:

- pyparsing
- cmd2

We thank the authors for writing this software and releasing them under the MIT license.

Finally, in order to generate movies through the Python library, you need to have ffmpeg installed and invokable from the command line. You do not need it to compile the code or to use it if you are not generating movies.

2.2 Linux

All of these programs/libraries can be found in most standard Linux installations, and if not, will be available to download from most package managers (e.g. apt-get, yum, pacman).

2.3 Mac OS X

For Mac users, all programs can be installed with either fink, MacPorts or homebrew. It is preferable that all are downloaded with just the one package manager in order to ensure they are compatible and function together correctly. Other options for installation are the Anaconda or Enthought Python distributions. A good reference which compares the available possibilities is the Python4Astronomers website.

2.4 Command-line code compilation

To compile GANDALF only as a C++ executable to be run from the command line without visualisation via the python program, first set the chosen C++ compiler in the Makefile (see section 3 for a full list of the variables you can change) by setting the variable `CPP` (if you leave it blank, you will get the default c++ compiler as defined by your system) and type

```
make executable
```

The code is compiled and linked with the chosen C++ compiler and all python components are ignored. If you do not have python installed on your system, or are having trouble getting the python components to function correctly, then the C++ executable can still be compiled and run stand-alone. Differently from other codes, in GANDALF we adopt the philosophy that the code should be compiled only once. If you need to change parameters, you can do it by setting them via the parameter files; in most cases no recompilation is needed. For this reason, if your physical set up does not require modifying the code, it could be that you need to compile GANDALF only once. If that is the case, you might want to copy the executable to one of your system directories. In this case, the command

```
make install
```

will copy the executable (provided you have already compiled the code) to your `/usr/local/bin` folder. In this way you should be able to invoke GANDALF from the command line just typing `gandalf` (i.e. without the need to specify the full path). Note that you might need to be root to write to that folder (in this case, use the sudo command).

2.5 Python code compilation

To compile all code elements including the python components, then complete the following steps :

- Install all python-related programs listed in Section 2
  - For Linux machines, python is usually installed by default. The additional python libraries should be easily installed using either the main package manager, `pip` or `easy_install`.
  - For Mac OS X, python is installed by default. However, it does not have directly compatible versions of all the required libraries. Therefore, it is required to install an additional version of python 2.7 using a 3rd party package manager like fink, Macports or homebrew.
- Set the required version of python in your Makefile (See Section 3). Since operating systems usually have more than one version of python installed, it is important to ensure that `make` uses the correct version when compiling the code. This is set using the `PYTHON` variable in the Makefile.

  Also, `make` requires the location of the python and numpy libraries. In most cases, `make` will be able to locate these libraries automatically (see description of Makefile options in section 3). However, if there is a problem, or you wish to use an alternative version of these libraries installed elsewhere on your system, then these can be set by the `PYLIB` and `NUMPY` variables in the Makefile.
- To compile all components of the code, including the python libraries with swig, type

  `make` or (equivalently) `make all`

- You now need to tell python how to find GANDALF when you want to use it. You have two options:
  - Add the location of the folder containing the main GANDALF directory to the `PYTHONPATH` environment variable. If you are using bash or related shells, then add the line

    ```
    export PYTHONPATH=XXX/YYY:$PYTHONPATH
    ```

    to your `.bashrc` (or `.bash_profile` on a Mac) script where `XXX/YYY/gandalf` is the absolute path of the GANDALF directory (note that you must not include "gandalf" in the folder name!). If you are using csh, tcsh or related shells, then add

```
setenv PYTHONPATH ‘‘XXX/YYY:$PYTHONPATH’’
```

to your relevant shell configuration file. Remember that you have to close and reopen the shell for this change to take effect, or to `source` the shell configuration file. In this case, python will use the version of GANDALF present in the folder you specified. Note that this *requires* the folder to be called `gandalf`; if you give another name, python will not be able to find GANDALF.

- If you know you will not be making changes to GANDALF, consider installing GANDALF in one of your system folders. In this case, just type

```
make installpython
```

and the GANDALF python library will be installed (this requires that you have already used `make` to compile the code). In this case, there is no need to set the variable `PYTHONPATH`. Also, the folder where you keep your original code does not need to have any special name. If later you do make changes to the code, remember to re-run this command - otherwise the installed version remains the old one! Note that you might need to be root (or to use sudo) to run this command.

3 Makefile options

The GANDALF Makefile is used to select options which are used to compile the code with. If the user wishes to change any compile-time options, the code must be recompiled from scratch by typing `make clean` and then `make`. For users of SEREN, the GANDALF Makefile has been simplified with many options either not present (since various specialist algorithms have not been implemented) or have been transferred to the parameters file. This has been done in order to make it less likely that the wrong Makefile options are used in simulations, and also to stop the need to recompile the code completely so often when using slightly different options.

Note that the Makefile in the root folder is just a convenience to set the variables you need - the actual Makefile is contained in the `src` folder.

- CPP : C++ compiler. If left blank (default), uses the default defined by your system. Depending on the values set here, we try to guess the compiler flags for your system. If you want to use a compiler different from the ones we support, you need to manually set the flags in the Makefile in the `src` folder. Note also that when using MPI we assume that you use g++/clang. If this is not the case, you need to manually edit the compiler flags in the Makefile in the `src` folder.
  g++ : GCC C++ compiler
  icpc : Intel C++ compiler
  clang : CLANG compiler
  mpic++ : Compiles the code with MPI.

  As an alternative, if you do not have the MPI compiler wrappers available (mpic++) you can enable MPI by adding the line

  ```
  MPI=1
  ```

  to the makefile. However, be warned that you may need to specify the location of the MPI header file, mpi.h, for compilation to succeed. You can do this by adding a line of the following form.

  ```
  INCLUDE = -I/path/to/mpi_header_directory
  ```

  Similarly, you may find that the default paths to the MPI library (-lmpi -lmpi++) may need to be changed in src/Makefile. **Where possible we highly recommend using the MPI compiler wrapper, mpic++.**
- PYTHON : name of python command-line executable (e.g. python, python2.7)
- COMPILER_MODE : Set compiler flags for production or debug runs
  DEBUG : Set all debug compiler options, including flags to use gdb debugger and full warning output (-Wall)
  STANDARD : Standard optimisation options (-O3)
  FAST : -O3 + fast flag options. Uses 'potentially unsafe' fast maths optimisation.
- PRECISION : Floating point precision
  SINGLE : 32-bit precision floating point variables
  DOUBLE : 64-bit precision floating point variables
- PYSNAP_PRECISION: Precision when using the python library (see section 8). The values returned by python will be in either single or double precision, depending on the value selected here. Note that this has nothing to do with the precision of the snapshots written to the disc! Most of the times one wants to match the precision used in the calculation. But to reduce the memory footprint required in the analysis, one might want to read the data in single precision even if it is stored in double precision on the disc.
  SINGLE : 32-bit precision floating point
  DOUBLE : 64-bit precision floating point
- OPENMP : Activate OpenMP directives during compilation (0 or 1)
- OUTPUT_LEVEL : Amount of output produced by code
  0 : No additional output
  1 : Minimal output of main-loop routines
  2 : Code routine marker output for all steps

- DEBUG_LEVEL : Amount of extra debug checking done by code
  0 : No extra debugging computations and output
  1 : Activate assert debug statements in code
  2 : Activate extra (expensive) debugging computations in code
- FFTW : Include FFTW (Fast Fourier transform) library for initial conditions (0 or 1). The FFTW_LIBRARY and FFTW_INCLUDE variables should contain the library links and include directory if different from the standard Linux directories (otherwise leave blank).
- GSL : Include GSL (GNU Scientific library) which is required for Ewald forces (0 or 1). The GSL_LIBRARY and GSL_INCLUDE variables should contain the library links and include directory if different from the standard Linux directories (otherwise leave blank).
- PYLIB : Path to directory that includes prefered python libraries. If left blank, `make` will use python routines to locate the libraries automatically.
- NUMPY : Path to directory that includes prefered numpy libraries. Ase with PYLIB, If left blank, `make` will use python routines to locate the library locations automatically.
- GTEST : Path to directory containing Google test suite library for unit testing (Currently set witu environment variable)

4 Basic usage

GANDALF can be run in three principle modes.

- Command-line mode, where the code is run as a C++ executable via the command line with a parameters file that selects which code options are used,
- via a python script where the code is run in a python environment with various plotting options available as well as running the code,
- in interactive mode where all the code and plotting options can be run directly by typing them into an interactive python shell.

4.1 Command-line mode

To run a simulation with the C++ executable on the command line, type :

```
./gandalf PARAMSFILE
```

where PARAMSFILE is the name of the parameters file for that simulation. If that parameters file does not exist in the current directory, or contains invalid parameter options, then the program will quit citing an error message. If the parameters file exists, then the code will parse the file, set-up the simulation and then run to completion.

4.2 Running python scripts

Any python script using the GANDALF python API can be run directly from the command-line as any regular python script. To run the script PYTHONSCRIPT.py, type

```
python PYTHONSCRIPT.py
```

or, depending on the default version of python on your system (e.g. installing matplotlib with fink on Mac OS X),

```
python2.7 PYTHONSCRIPT.py
```

The code will remain inside the python environment until the script is fully executed. Python typically checks that the entire file has valid syntax before running each command.

4.3 Interactive mode

To open the interactive viewer, type :

```
python analysis/gandalf_interpreter.py
```

or, depending on the default version of python on your system (e.g. installing matplotlib with fink on Mac OS X),

```
python2.7 analysis/gandalf_interpreter.py
```

The code should open with a splash screen containing the code title followed by a command prompt of the form '`gandalf >`'. To run a simulation defined by a parameter file, then type

```
newsim PARAMSFILE
```

and then

```
run
```

The current simulation can be plotted at any point by using simple commands such as, for example,

```
plot x y
```

For a list on available commands, type `help` in the command line. For more detailed information on the functionality of a particular command, type `help command`. For more information on interactive python commands, see Section 8.

5 Parameter options

With the exception of a few compiled-time options that are selected in the Makefile, all physics and code feature options are controlled from the parameters file, including the dimensionality, simulation type and SPH options. We list here all possible parameters, including all possible options for those parameters with a limited choice. We note that a parameters file need not contain a value for each parameter, in which case the default value is taken.

5.1 Main simulation parameters

- `ndim` : Simulation dimensionality (1, 2 or 3)
- `sim` : Simulation type
  - sph = SPH (+ N-body) algorithm (default : 'grad-h' SPH)
  - gradhsph = 'grad-h' SPH simulation (+ N-body)
  - sm2012sph = Saitoh & Makino (2012) SPH (+ N-body)
  - meshlessfv = Meshless Finite-Volume algorithm (default : 'mfvmuscl')
  - mfvmuscl = Meshless FV MUSCL integration simulation
  - mfvrk = Meshless FV Runge-Kutta integration
  - nbody = N-body only simulation
- `nbody` : Main N-body integration algorithm
  - lfkdk = 2nd-order Leapfrog kick-drift-kick
  - lfdkd = 2nd-order Leapfrog drift-kick-drift
  - hermite4 = 4th-order Hermite scheme
  - hermite4ts = Time-symmetric 4th-order Hermite scheme
- `ic` : Simulation initial conditions
  - file = Load initial conditions from external file
  - bb = Boss-Bodenheimer (1979) test
  - binary = Simple binary star test
  - binaryacc = Binary accretion test
  - blastwave = Blastwave test
  - bondi = Spherically symmetric Bondi accretion test
  - box = Create a uniform box of gas
  - cdiscontinuity = Contact discontinuity test
  - disc = Accretion disc initial conditions
  - ewaldcylinder = Cylinder for 1D Ewald gravity test
  - ewaldsine = Sinusoidal density for 3D Ewald gravity test
  - ewaldslab = Slab for 2D Ewald gravity test
  - gresho = Gresho-Chan vortex test
  - khi = Kelvin-Helmholtz instability test
  - noh = Noh problem initial conditions
  - plummer = Plummer sphere test
  - quadruple = Simple hierarchical quadruple star test
  - sedov = Sedov blast-wave test
  - shearflow = Shear flow test
  - shocktube = Shocktube test
  - soundwave = 1D soundwave perturbation
  - sphere = Uniform density sphere
  - spitzer = Ionised bubble Spitzer expansion test
  - triple = Simple hierachical triple star test
  - turbcore = Turbulent spherical, self-gravitating core
  - python = Generate initial conditions from python
- `run_id` : Simulation run id string

- `in_file` : Input filename (when `ic = file`)
- `in_file_form` : Format of initial conditions file
  column = Simple column data format
  sf/seren_form = SEREN ASCII format
  su/seren_unform = SEREN binary format
- `out_file_form` : Format of outputted snapshot files
  column = Simple column data format
  sf/seren_form = SEREN ASCII format
  su/seren_unform = SEREN binary format
- `tend` : Termination time of the simulation (given in `tunits`)
- `tmax_wallclock` : Maximum allowed wallclock time for simulation before being terminated
- `dt_snap` : Snapshot time interval (given in `tunits`)
- `tsnapfirst` : Time of first snapshot (given in `tunits`)
- `Nstepsmax` : Maximum no. of steps in simulation before termination
- `noutputstep` : Frequency of screen output (in units of integer steps)
- `ndiagstep` : No. of complete block steps between diagnostic output
- `nrestartstep` : No. of full block steps before producing restart dump
- `litesnap` : Output 'lite' snapshots (for generating movies)? (0 or 1)
- `dt_litesnap` : Lite snapshot time interval (given in `tunits`)
- `tlitesnapfirst` : Time of first lite snapshot (given in `tunits`)

5.2 Unit parameters

- `dimensionless` : Are all quantities dimensionless? (0 or 1)
- `routunit` : Position unit
  pc/kpc/mpc = parsec/kiloparsec/megaparsec
  au = astronomical unit
  r_sun = Solar radius
  r_earth = Earth radius
  cm/m/km = centimetre/metre/kilometre
- `moutunit` : Mass unit
  m_sun = Solar mass
  m_jup/m_earth = Jupiter mass/Earth mass
  g/kg = gram/kilogram
- `toutunit` : Time unit
  yr/myr/gyr = year/megayear/gigayear
  day = day
  sec = second
- `voutunit` : Velocity unit
  cm_s/m_s/km_s = centimetres/metres/kilometres per second
  au_yr = astronomical units per year
- `aoutunit` : Acceleration unit
  cm_s2/m_s2/km_s2 = cm/m/km per second squared
  au_yr2 = astronomical units per year squared

- `rhooutunit` : Density unit
  m_sun_pc3 = Solar masses per parsec cubed
  kg_m3 = kilogrammes per metre cubed
  g_cm3 = grammes per centimetre cubed
- `sigmaoutunit` : Column/surface density unit
  kg_m2 = kilogrammes per meter squared
- `pressoutunit` : Pressure unit
  Pa = pascals/newtons per square metre
  bar = bars
- `foutunit` : Force unit
  N = newtons
  dyn = dynes
- `Eoutunit` : Energy unit
  J/GJ = joules/gigajoules
  erg = ergs
  eV = electron volts
  10$\hat{4}$0erg = 10^{40} ergs
- `momoutunit` : Momentum unit
  m_sunkm_s = Solar masses kilometres per second
  m_sunau_yr = Solar masses A.U. per year
  kgm_s = Kilogram metres per second
  gcm_s = Gram centimetres per second
- `angmomoutunit` : Angular momentum unit
  m_sunkm2_s = Solar masses kilometres squared per second
  m_sunau2_yr = Solar masses A.U. squared per year
  kgm2_s = Kilogram metres squared per second
  gcm2_s = Gram centimetres squared per second
- `angveloutunit` : Angular velocity unit
  rad_s = Radians per second
- `dmdtoutunit` : Mass (accretion) rate unit
  m_sun_myr = Solar masses per megayear
  m_sun_yr = Solar masses per year
  kg_s = kilogrammes per second
  g_s = grammes per second
- `Loutunit` : Luminosity unit
  L_sun = Solar luminosity
  W = watts
  erg_s = ergs per second
- `kappaoutunit` : Mass opacity unit
  m2_kg = metre squared per kilogram
  cm2_g = centimetre squared per gram
- `Boutunit` : Magnetic field unit
  tesla = tesla
  gauss = gauss
- `Qoutunit` : Charge unit
  C = coulomb
  e = electron charge
- `Jcuroutunit` : Current density unit
  C_s_m2 = coloumb per second per metre squared

- `uoutunit` : Specific energy unit
  J_kg = Joules per kilogram
  erg_g = ergs per gram
- `dudtoutunit` : Heating rate unit
  J_kg_s = Joules per kilogram per second
  erg_g_s = ergs per gram per second
- `tempoutunit` : Temperature unit
  K = Kelvin

5.3 Integration and timestep parameters

- `accel_mult` : Acceleration timestep multiplier
- `courant_mult` : Courant timestep multiplier
- `visc_mult` : Viscosity timestep multiplier
- `nbody_mult` : N-body timestep multiplier
- `subsys_mult` : Sub-system N-body timestep multiplier
- `Nlevels` : No. of initial timestep levels
- `level_diff_max` : Maximum allowed SPH neighbour timestep difference
- `sph_single_timestep` : Constrain all SPH particles to a single timestep level
- `nbody_single_timestep` : Constrain all N-body particles to a single timestep level

5.4 Hydrodynamical parameters

- `hydro_forces` : Compute hydro forces? (1 or 0)
- `gas_eos` : Gas particles equation-of-state
  energy_eqn = Solve energy equation
  isothermal = Isothermal EOS
  barotropic = Barotropic EOS (i.e. for mimicing isothermal + adiabatic phase during protostellar collapse)
  barotropic2 = Similar to barotropic, but using discrete power laws rather than smooth change
  rad_ws = EOS relating to Stamatellos et al. (2007) cooling method
  disc_locally_isothermal = Locally isothermal equation of state, to be used with the disc setup.
- `energy_integration` : Energy integration scheme (only applicable if solving the energy equation)
  null = Energy equation not integrated separately
  rad_ws = Integrate energy terms using Stamatellos et al. (2007) method
- `energy_mult` : Explicit energy integration timestep multiplier
- `gamma_eos` : Ratio of specific heats for gas
- `temp0` : (Isothermal) temperature (isothermal or barotropic EOS)
- `mu_bar` : Mean gas particle mass (in units of hydrogen mass)
- `rho_bary` : Adiabatic density turnover in barotropic EOS (in g/cm$\hat{3}$)
- `eta_eos` : Polytropic exponent (for barotropic EOS)
- `radws_table` : Name of EOS file for Stamatellos et al. (2007) cooling method
- `temp_ambient` : Ambient temperature (for rad_ws method)
- `lombardi_method` : Use the Lombardi et al. (2015) metric for rad_ws method

5.5 SPH parameters

- `sph_integration` : SPH particle integration scheme
  lfkdk = 2nd-order Leapfrog kick-drift-kick
  lfdkd = 2nd-order Leapfrog drift-kick-drift
- `kernel` : SPH kernel function
  m4 = M4 Cubic spline kernel
  quintic = Quintic spline kernel
  gaussian = Gaussian kernel (truncated at 3h)
- `tabulated_kernel` : Tabulate kernel function (1 or 0)
- `h_fac` : Particles-per-smoothing length factor (eta in papers)
- `h_converge` : Smoothing length iteration convergence tolerance

5.6 Artificial viscosity parameters

- `avisc` : Artificial viscosity options
  none = No artificial viscosity
  mon97 = Monaghan (1997) viscosity
- `acond` : Artificial conductivity options
  none = No artificial conductivity
  price2008 = Price (2008) conductivity
  wadsley2008 = Wadsley et al. (2008) conductivity
- `time_dependent_avisc` : Aritificial viscosity switch
  none = No switch, artificial viscoisty always set to `alpha_visc`.
  mm97 = Morris & Monaghan (1997) switch
  cd2010 = Advanced switch by Cullen & Dehnen (2010), using the balsar swith
- `alpha_visc` : (Maximum) value of alpha viscosity parameter
- `alpha_visc_min` : Minimum value of alpha for time-dependent viscosity
- `beta_visc` : Value of beta viscosity as a multiple of alpha

5.7 Meshless finite-volume parameters

- `riemann_solver` : Riemann solver in FV scheme
  exact = Exact Riemann solver (e.g. Toro 1999)
  hllc = HLLC approximate Riemann solver
- `slope_limiter` : Slope limiter for TVD condition
  null = No limiting
  zeroslope = Set all slopes to zero (effectively 1st order Godunov)
  balsara2004 = Balsara (2004) slope-limiter
  springel2009 = Original AREPO (Springel 2009) slope limiter
  tess2011 = TESS slope limtier
  gizmo = Original GIZMO paper (Hopkins 2015) slope limiter
  minmod = simplified implementation of minmod slope limiter
- `zero_mass_flux` : Use Meshless-Finite Mass (MFM) scheme to prevent mass-flux between particles? Should be activated if self-gravity or star/sink particles are used in conjunction with any MFV scheme. (1 or 0)
- `static_particles` : Use static particles (Eulerian approach)? (1 or 0)

- `time_step_limiter` : Use an additional limiter on the time-steps
  none = No additional limiter
  conservative = Predictive global timestep limiter of Springel (2009)
  simple = Saitoh & Makino type limiter. Reduces the time step if it is detected to be too large.
- `shear_visc` Enable viscosity with constant kinematic shear viscoity
- `bulk_visc` Enable viscosity with constant kinematic bulk viscoity

5.8 Gravitational parameters

- `self_gravity` : Compute gravitational forces? (1 or 0)
- `kgrav` : Direction of (external) gravitational acceleration (0, 1 or 2)
- `grav_kernel` : Form of gravitational softening
  mean_h = Mean smoothing length softening
- `external_potential` : External gravitational potential
  none = No external potential
  vertical = Constant gravitational field
  plummer = Plummer background potential
- `avert` : Veritcal (constant) gravitational acceleration
- `rplummer_extpot` : Background Plummer potential radius
- `mplummer_extpot` : Background Plummer potential mass

5.9 Neighbour searching and tree parameters

- `neib_search` : Neighbour searching algorithm
  bruteforce = Brute-force (i.e. summation over all particles)
  kdtree = Balanced kd-binary tree
  octtree = Barnes-Hut octal tree
- `gravity_mac` : Gravity-tree cell-opening criteria (N.B. always defualts to geometric for now). The eigenmac and gadget2 criteria are more accurate for a given computational cost.
  geometric = Standard Barnes-Hut geomtric opening angle criterion. This can fail in pathalogical cases, such as when the forces nearly balance.
  eigenmac = Compute eigenvalues of quadrupole moment tensor for MAC (Hubber et al. 2011)
  gadget2 = Relative opening criterion based on comparing estimate of force error to the previous accleleration (Springel 2005).
- `multipole` : Multipole expansion for tree-gravity
  monopole = Monopole-only terms for cell gravity
  quadrupole = Include quadrupole moment terms for cell gravity
  fast_monopole = Compute monpoles more efficiently using Taylor expansion about cell COM
- `Nleafmax` : Maximum no. of particles allowed in tree leaf cell
- `ntreebuildstep` : Integer steps inbetween tree re-builds
- `ntreestock` : Integer steps inbetween tree re-stocks
- `thetamaxsqd` : Maximum tree gravitational walk opening angle (squared)
- `macerror` : MAC error tolerance for individual cells

5.10 N-body parameters

- `sub_systems` : Identify and integrate sub-systems separately? (0 or 1)
- `sub_system_integration` : Main N-body integration algorithm
  lfkdk = 2nd-order Leapfrog kick-drift-kick
  hermite4 = 4th-order Hermite scheme
  hermite4ts = Time-symmetric 4th-order Hermite scheme
- `Npec` : No. of P(EC)^n iterations in time-symmetric scheme (if non time-symmetric scheme is used, automatically sets to 1)
- `nbody_softening` : Use SPH kernel-softening between star particles? (0 or 1)
- `binary_stats` : Output binary statistics? (1 or 0)
- `nsystembuildstep` : Integer steps inbetween re-building the sub-system tree.
- `gpefrac` : Maximum fraction of total gravitational potential energy from external sources to allow sub-system.

5.11 Sink particle parameters

- `sink_particles` : Do stars/sinks accrete? (0 or 1)
- `create_sinks` : Create new sink particles? (0 or 1)
- `smooth_accretion` : Use smooth accretion? (0 or 1)
- `fixed_sink_mass` : Fixed sink mass, even when accreting? (0 or 1)
- `extra_sink_output` : Extra output of sink particles? (0 or 1)
- `rho_sink` : Sink particle creation density (in cgs units)
- `alpha_ss` : Sunyaev-Shakura alpha for smooth disc accretion
- `sink_radius` : Sink particle radius (in units of smoothing length)
- `smooth_accrete_frac` : Smooth accretion instantaneous accretion mass frac.
- `smooth_accrete_dt` : Smooth accretion instantaneous accretion timestep frac.
- `sink_radius_mode` : How to calculate new sink radius
  hmult = sink radius a multiple of SPH particle smoothing length
  fixed = sink radius is fixed for all new sinks

5.12 Radiation parameters

- `radiation` : Main radiation algorithm used
  none = No radiation field
  ionisation = Multiple source ionising radiation
- `Nphoton` : No. of photon packets (for Monte-Carlo radiation transport)
- `mu_ion` : Mean-gas particle mass for ionised gas
- `temp_ion` : Temperature of ionised gas
- `arecomb` : Recombination coefficient (in cgs units)
- `Ndotmin` : No. of ionising photons per second
- `NLyC` : No. of ionising photons per second

5.13 Radiative feedback parameters

- `rad_fb` : Turn on radiative feedback for supported specific internal energy integration methods.
- `ambient_heating` : Add heating from an ambient background defined by `temp_ambient`
- `disc_heating` : Add heating from a temperature profile with a central star (1) or binary (2)
- `sink_heating` : Add accretion heating from sinks (except central objects from `disc_heating`)
- `sink_fb` : Type of heating from sinks (except central objects)
  continuous = Heating from accretion as mass is accreted
  episodic = NOT IMPLEMENTED
- `r_smooth` : Smoothing radius for `disc_heating` temperature profile
- `temp_q` : Power index for primary in `disc_heating` temperature profile
- `temp_q_secondary` : Power index for secondary in `disc_heating` temperature profile
- `temp_au` : Temperature at 1 AU for primary in `disc_heating` temperature profile
- `temp_au_secondary` : Temperature at 1 AU for secondary in `disc_heating` temperature profile
- `f_acc` : Fraction of energy converted to heating from accretion (use with `sink_heating`)
- `r_star` : Radius of star ($M_{sink} > 80\ M_{jup}$) in solar radii (use with `sink_heating`)
- `r_bdwarf` : Radius of brown dwarf ($13 > M_{sink} > 80\ M_{jup}$) in solar radii (use with `sink_heating`)
- `r_planet` : Radius of planet ($M_{sink} < 13\ M_{jup}$) in solar radii (use with `sink_heating`)

5.14 Boundary parameters

- `boundary_lhs[0]` : Boundary conditions for LHS of x-dimension
- `boundary_rhs[0]` : Boundary conditions for RHS of x-dimension
- `boundary_lhs[1]` : Boundary conditions for LHS of y-dimension
- `boundary_rhs[1]` : Boundary conditions for RHS of y-dimension
- `boundary_lhs[2]` : Boundary conditions for LHS of z-dimension
- `boundary_rhs[2]` : Boundary conditions for RHS of z-dimension For all boundaries:
  open = open boundaries (i.e. extends to infinity)
  periodic = periodic wrapping between LHS & RHS boundary
  wall = wall at boundary (i.e. reflection of particles)
- `boxmin[0]` : Location of LHS x-boundary
- `boxmax[0]` : Location of RHS x-boundary
- `boxmin[1]` : Location of LHS y-boundary
- `boxmax[1]` : Location of RHS y-boundary
- `boxmin[2]` : Location of LHS z-boundary
- `boxmax[2]` : Location of RHS z-boundary

5.15 Initial conditions parameters

- `particle_distribution` : Particle configuration when generating uniform density fluids on the fly
  random = Particle positions generated with random number generator
  cubic_lattice = Particle positions generated on a uniform cubic lattice
  hexagonal_lattice = Particle positions generated on a hexagonal closed-packed array
- `smooth_ic` : Smooth any particle quantities around discontinuities
- `com_frame` : Translate ICs to COM frame before starting simulation
- `Nhydro` : No. of hydrodynamical particles
- `Nhydromax` : Maximum no. of hysrodynamical particles
- `Nstar` : No. of star particles
- `Nstarmax` : Maximum no. of star particles
- `Nlattice1[0]` : No. of ptcls on lattice 1 in x-dimension
- `Nlattice1[1]` : No. of ptcls on lattice 1 in y-dimension
- `Nlattice1[2]` : No. of ptcls on lattice 1 in z-dimension
- `Nlattice2[0]` : No. of ptcls on lattice 2 in x-dimension
- `Nlattice2[1]` : No. of ptcls on lattice 2 in y-dimension
- `Nlattice2[2]` : No. of ptcls on lattice 2 in z-dimension
- `vfluid1[0]` : x-velocity of fluid 1
- `vfluid1[1]` : y-velocity of fluid 1
- `vfluid1[2]` : z-velocity of fluid 1
- `vfluid2[0]` : x-velocity of fluid 2
- `vfluid2[1]` : y-velocity of fluid 2
- `vfluid2[2]` : z-velocity of fluid 2
- `rhofluid1` : Density of fluid 1
- `rhofluid2` : Density of fluid 2
- `press1` : Pressure of fluid 1
- `press2` : Pressure of fluid 2
- `amp` : Amplitude of applied perturbation
- `lambda` : Wavelength of applied perturbation
- `kefrac` : Fraction of energy that is kinetic (Sedov test)
- `radius` : Radius of cloud
- `angvel` : Angular velocity of cloud (in radians per second)
- `mcloud` : Mass of cloud
- `rplummer` : Plummer radius
- `mplummer` : Total mass of plummer sphere
- `rstar` : (Softening) radius of star particles
- `cdmfrac` : Fraction of mass in cdm particles
- `gasfrac` : Fraction of mass in gas particles

- `starfrac` : Fraction of mass in star particles
- `m1` : Mass of star 1
- `m2` : Mass of star 2
- `m3` : Mass of star 3
- `m4` : Mass of star 4
- `abin` : Semi-major axis of binary orbit 1
- `abin2` : Semi-major axis of binary orbit 2
- `ebin` : Orbital eccentricity of binary orbit 1
- `ebin2` : Orbital eccentricity of binary orbit 2
- `phirot` : Phi Euler rotation angle
- `thetarot` : Theta Euler rotation angle
- `psirot` : Psi Euler rotation angle
- `vmachbin` : Speed of binary COM through ambient gas
- `alpha_turb` : Turbulent energy (as multiple of gravitational energy)
- `power_turb` : Power spectrum slope of initial turbulent velocity field
- `asound` : Sound speed
- `zmax` : ??
- `DiscIc`: There are multiple parameters prefixed with this string; they contol the accretion disc setup. Please refer to section 7.2.1 for an extensive explanation.

5.16 Regularised initial conditions parameters

- `regularise_particle_ics` : Regularise particle initial conditions before main simulation
- `regularise_smooth_density` : Regularisation uses the smoothed density field
- `Nreg` : No. of regularisation iterations
- `alpha_reg` : Glass-creation regularisation factor (0 = no glass; > 0 = glass-like)
- `rho_reg` : Density-field regularisation factor (> 0 = iterates to required density field)
- `Nreg` : No. of regularisation iterations

5.17 Random number generator parameters

- `rand_algorithm` : Random number generator algorithm
  none = No algorithm selected; use intrinsic generator
  (not recommended since this is system dependent and is not reproducable on different machines)
  xorshift = Xorshift generator (see Numerical recipes, Ed 3, Chapter 7 for details)
- `randseed` : Random number seed

5.18 MPI parameters

- `mpi_decomposition` : Mode of MPI decomposition
  kdtree = Use simple KD-tree decomposition
- `pruning_level_min` : Minimum level to prune exported trees
- `pruning_level_max` : Maximum level to prune exported trees

5.19 Python viewer parameters

- `dt_python` : Time interval (in seconds) between view window updates

6 Input and output file formats

GANDALF supports several simple file formats for reading in initial conditions or outputting snapshots for visualisation and analysis.

6.1 Column format

The column format is the simplest snapshot file format in GANDALF and consists of a simple header plus a fixed column-data format with particle data. It is designed for ease-of-use in quickly generating initial conditions, or perhaps porting initial conditions from other codes/generators. It is certainly not designed as a long-term format for use in large simulations, partly due to being an ASCII file which can quickly get unfeasibly large for big simulations.

The header comprises of four lines, each with one variable in the following order :

- `Nhydro` : No. of hydro particles
- `Nstar` : No. of star/sink particles
- `ndim` : Dimensionality of snapshot data
- `t` : Time of snapshot

Each of the 4 header variables is preceded by a hash ('#') in order that simple plotting programs (e.g. gnuplot) will ignore the headers allowing the particle data to be simply plotted. Older versions of GANDALF may not have this hash but nevertheless should still easily be read-in by the code.

The particle data is then split up into `Nhydro` + `Nstar` rows, hydro particles first and then star particles. The data columns are ordered as :

- `x [,y, z]` : Particle position vectors (depending on value of `ndim`)
- `vx [,vy, vz]` : Particle velocity vectors (depending on value of `ndim`)
- `m` : Particle mass
- `h` : Particle smoothing length (for stars also; radius = `2/3 h` depending on kernel)
- `rho` : Particle density (set to 0 for stars)
- `u` : Particle specific internal energy (set to 0 for stars)

Since the column format was only designed for simple IO and plotting, it is a rather minimalistic format that does not necessarily contain all the required data or options for full simulations. For example, all stars are automatically set to being sinks. Therefore it is not recommended for long-term use other than simple plotting (e.g. for debugging or developing/transfering ICs).

6.2 SEREN format

The SEREN format is a legacy format from the original SEREN code which is currently the main format for use in GANDALF. There are two different version of the SEREN format;

- `seren_form` or `sf` : 'Formatted' (i.e. ASCII)
- `seren_unform` or `su` : 'Unformatted' (i.e. binary)

The format consists of a large multi-part header with different data types, including information about the units used as well as the particle data and star/sink data arrays. The full header array sizes are given below. In the formatted/ASCII case, each array element is printed on a separate line but is compressed as a single stream. For any fields not given below, assume the quantity to be unused and therefore defaulted to zero in the header. The header reads as follows:

- `format_id` : A string identifying the exact format type (for verification)
- `pr` : Floating point precision (4 = single precision, 8 = double precision)
- `ndim` : Spatial dimensionality
- `vdim` : Velocity dimensionality
- `bdim` : Magnetic field dimensionality
- `idata[50]` : Integer variables
  - idata[0] = `Nhydro` : No. of hydro particles (SEREN = `pgas`)
  - idata[1] = `Nstar` : No. of star/sink particles (SEREN = `stot`)
  - idata[2] = N/A in GANDALF (SEREN = `pboundary` : No. of static boundary particles)
  - idata[3] = N/A in GANDALF (SEREN = `picm` : No. of 'inter-cloud medium' particles)
  - idata[4] = `Ngas` : No. of self-gravitating gas particles (SEREN = `pgas`)
  - idata[5] = `Ncdm` : No. of self-gravitating cdm particles (SEREN = `pcdm`)
  - idata[6] = `Ndust` : No. of dust particles (SEREN = `pdust`)
  - idata[7] = N/A in GANDALF (SEREN = `pion` : No. of particles)
  - idata[19] = `nunit` : No. of unit variables in header
  - idata[20] = `ndata` : No. of data arrays in snapshot file
  - idata[29] = N/A in GANDALF (SEREN = `dmdt_range` : No. of accretion rate variables in sink array)
  - idata[30] = N/A in GANDALF (SEREN = `pgas_orig` : Original no. of gas particles)
  - idata[31] = N/A in GANDALF (SEREN = `pp_gather` : Average/exact no. of neighbours)
  - idata[39] = N/A in GANDALF (SEREN = `rank` : MPI process that created the file)
  - idata[40] = N/A in GANDALF (SEREN = `Nmpi` : Total no. of MPI processes)
- `ilpdata[50]` : Long integer variables
  - ilpdata[0] = `Noutsnap` : No. of snapshot files created (SEREN = `snapshot`)
  - ilpdata[1] = `Nsteps` : No. of complete integration steps (SEREN = `nsteps`)
  - ilpdata[2] = N/A in GANDALF (SEREN = `ntempnext` : Integer time for next temporary snapshot)
  - ilpdata[3] = N/A in GANDALF (SEREN = `ndiagnext` : Integer time for next diagnostic output)
  - ilpdata[4] = N/A in GANDALF (SEREN = `nsnapnext` : Integer time for next integer snapshot)
  - ilpdata[5] = N/A in GANDALF (SEREN = `nsinknext` : Integer time for next sink output)
  - ilpdata[10] = `Noutlitesnap` : No. of lite snapshots (N/A in SEREN)
- `rdata[50]` : Standard precision floating-point variables
  - rdata[0] = `h_fac` : 'No. of particles per smoothing length' factor in h-rho iteration (SEREN = `h_fac`)
  - rdata[1] = N/A in GANDALF (SEREN = `gamma` : Ratio of specific heats for gas)
  - rdata[2] = N/A in GANDALF (SEREN = `mu_bar` : Mean gas particle mass (in units of m_h))
  - rdata[3] = N/A in GANDALF (SEREN = `hmin` : Minimum smoothing length)
- `ddata[50]` : Double precision floating-point variables
  - ddata[0] = `t` : Simulation time when snapshot was created (SEREN = `time`)

  - ddata[1] = `tsnaplast` : Time that previous snapshot was created (SEREN = `lastsnap`)
  - ddata[2] = `mmean` : Average mass of gas particles (SEREN = `mgas_orig`)
  - ddata[10] = `tlitesnaplast` : Time when previous lite snapshot was created (N/A in SEREN)
- `unit_data[nunit]` : Strings identifying the chosen input unit for each physical quantity. Note that this does not necessarily have to be the same as the chosen output units (See Section 11). In unformatted/binary files, the strings are set to a default maximum length of 20 characters (with white space padding at the end).
- `data_id[ndata]` : Strings identifying the physical quantities contained in the particle arrays that follow the header. Note that vector quantities are written grouping together the different components for each component, rathern than different ndim scalar values (e.g., in 2d x and y of a given particle are written one after the other, and after come the x and y of another particle). As with the `unit_data` array using unformatted/binary file, the strings are set to a default maximum length of 20 characters (with white space padding at the end).
  - `porig` : Original (unique) particle ids
  - `r` : Hydro particle positions
  - `m` : Hydro particle masses
  - `h` : Hydro particle smoothing lengths
  - `v` : Hydro particle velocities
  - `rho` : Hydro particle densities
  - `u` : Hydro particle specific internal energies
  - `sink_v1` : Star/sink particle data (N.B. a more complicated data structure due to the various star/sink properties in SEREN; recommended to look at the source code for more info)
- `typedata` : Information on each particle array, such as the variable type, unit type, etc.. For the data held in array `idata`, the data information in each field is :
  - typedata[idata][0] : Dimensionality of array (i.e. 1 to ndim)
  - typedata[idata][1] : i.d. of first particle in array in Fortran (i.e. normally 1 as opposed to 0 in C/C++)
  - typedata[idata][2] : i.d. of last particle in array in Fortran (i.e. normally Nhydro as opposed to Nhydro - 1 as in C/C++)
  - typedata[idata][3] : Variable type of array (1 : bool; 2 : integer; 3 : long integer; 4 : float; 5 : double)
  - typedata[idata][4] : Physical unit of array (same i.d. as `unit_data` array index plus one due to Fortran array convention)

The headers, in particular `data_id` and `typedata`, contain all the information about what type of data is contained in the particle arrays. The particle arrays come immediately after the end of the header.

6.3 SEREN 'lite' format

The SEREN 'lite' format is a minimised version of the SEREN format designed for creating many snapshots for movies. It has various restrictions which do NOT make it useful for other forms of analysis or restarting simulation, such as :

- No velocity information (only contains position, mass, smoothing length, density and specific internal energy)
- Hard-wired to single precision to obtain the smallest file-size in binary format

It can used simulataneously with the other more complete formats, which can instead be used for the analysis or for restarting. There are three parameters which control the usage of the lite formats, `litesnap`, `dt_litesnap` and `tlitesnapfirst` (See parameter tables)

7 Generating initial conditions

There are three main ways of generating initial conditions in GANDALF.

7.1 'On-the-fly' initial conditions

It is possible to generate initial conditions 'on-the-fly' using internal subroutines in GANDALF. At present, the following initial conditions are included in the code :

Hydrodynamical simulations :

- bb : Boss-Bodenheimer test
- binaryacc : Binary accretion simulation
- blastwave : 1D blastwave test
- bondi : Spherically symmetric Bondi accretion test
- box : Create uniform box
- cdiscontinuity : Contact-discontinuity test
- disc : Accretion disc. See section 7.2.1.
- ewald : Several tests of Ewald gravity
- gresho : Gresho vortex test
- khi : Kelvin-helmholtz instability
- noh : Noh shock test
- plummer : Plummer sphere (stars + gas, or just gas)
- sedov : Sedov blastwave test
- shearflow : Shearing flow test
- shocktube : Simple two-fluid shocktube test
- soundwave : Simple 1D sound-wave perturbation test
- sphere : Create uniform density sphere
- spitzer : Spitzer expansion of HII region test
- turbcore : Create uniform density core with turbulent velocity field

N-body simulations :

- binary : Simple circular binary system test
- burrau : Burrau Pythagorean test
- figure8 : Simple 3-body figure-8 test
- plummer : Plummer sphere (stars + gas, or just stars)
- quadruple : Simple hierarchical quadruple system test
- triple : Simple hierarchical triple system test

7.1.1 Creating your own initial conditions generators

The subroutines that create the initial conditions are all contained in the `Ic` class. This class is essentially a container for all subroutines and helper functions in one place. If the user wishes to create their own initial conditions subroutines inside GANDALF, then there are 3 important files that must be editted.

- `src/Headers/IC.h` : this file contains the class definition for the `Ic` class. The function prototype for any new IC generating function should be placed in this class, e.g.

  ```
  void MyNewInitialConditions(void);)
  ```

- `src/Common/Ic.cpp` : this file contains all the functions contained in the Ic class, including the helper functions. The full code for generating the initial conditions should be placed in this file, e.g.

  ```
  void Ic::MyNewInitialConditions(void) {....}
  ```

- `src/Common/SimulationIC.hpp` : Interface file that links (and calls) the IC class functions from the main simulation class. In this file, you will need to add your own function call (together with the 'if' statement to call the function), e.g.

  ```
  else if (ic == ''myic'') {icGenerator.MyNewInitialConditions();}
  ```

To select your own initial conditions when running GANDALF, you simply set the `ic` parameter in your parameters file to the chosen string, e.g. `ic = myic`.

7.2 Description of initial conditions

7.2.1 Accretion disc

If "ic" is set to "disc", a standard thin accretion disc (e.g., **?**) will be generated. See table 1 for a summary of the parameters. This setup can be used in 2-d (that is, vertically integrated) or in 3-d. Currently the setup only works in dimensionless units. A star with a mass of 1 is created at the origin of the coordinate system. The disc will extend in radius from "DiscIcRin" to "DiscIcRout". The disc has an initial power-law surface density profile $\Sigma \propto r^{-p}$, where the value of the exponent p is given by the parameter "DiscIcP". The normalization is fixed by the parameter "DiscIcMass", which is the total mass of the disc (relative to the star). This setup is designed to use a locally isothermal equation of state, where the sound speed follows a power-law with radius: $c_s \propto r^{-q}$, where the exponent q is given by the parameter "DiscIcQ". This equation of state is selected if "gas_eos" has the value "disc_locally_isothermal". The setup emits a warning if one uses a different equation of state. When running in 3-d, the disc is assumed to be vertically isothermal and in hydrostatic equilibrium in the vertical direction, that is, $\rho \propto \exp(-z^2/2H^2)$, where $H = c_s/\Omega$ and Ω is the keplerian angular frequency. The normalization of the sound speed is set by the parameter "DiscIcHr", which sets the aspect ratio H/r at the inner radius of the disc. Notice that we use random placement to initialise the disc; the initial density structure will thus be noisy. This in general not an issue since the particles typically rearrange themselves over one orbit. However, please keep this into account if you a need a structure with little noise already in the initial conditions.

The setup also supports the option of embedding a planet in the disc. This is selected if the parameter "DiscIcPlanet" takes a value of 1. The planet will be on a keplerian orbit; see table 7.2.1 for setting the orbital parameters and planet mass. The setup does not currently support multiple planets, but it would be very straightforward to extend it - please contact the authors if you are interested in it.

As explained in chapter 9, GANDALF can solve for the dust dynamics in addition to the gas dynamics. If you wish to follow also the dust dynamics, the parameter "NDust" controls the number of dust particles and the parameter "DustGasRatio" gives the mass ratio between the dust and the gas. Notice that when using dust the parameter "DiscIcMass" refers to the *total* (i.e., gas+dust) mass of the disc. Other choices of the dust algorithm are described in chapter 9.

Finally, the star and the planet will accrete mass from the disc if sink particles are switched on in the disc. This is controlled by the normal parameter "sink_particles". The accretion radius of the star is always fixed to the inner radius of the disc; the accretion radius of the planet instead is controlled by the parameter "DiscIcPlanetAccretionRadiusHill", which sets the accretion radius in units of the Hill radius of the planet. Notice that, if the planet accretes, effectively its gravity is not smoothed, since we smooth the gravitational potential only inside the accretion radius. If you are not using sink particles, instead, the parameter "DiscIcPlanetAccretionRadiusHill" expresses the smoothing length of the gravitational potential. Currently is not possible to let only the star accrete and not the planet (or viceversa); please contact the authors if you require this behaviour.

Table 1 lists all the parameters discussed so far and their default values. Most of the default values come from the standard setup of **?**; note however that we use a surface density $\propto r^{-1}$ rather than a flat one. We also do not employ ghost particles at the inner and outer boundary.

Parameter name	Description	Default value
DiscIcMass	Disc mass	0.01
DiscIcP	Power-law exponent of the surface density	1
DiscIcQ	Power-law exponent of the sound speed	0.5
DiscIcRin	Inner radius of the disc	0.4
DiscIcRout	Outer radius of the disc	2.5
DiscIcHr	Aspect ratio of the disc at the inner radius	
DiscIcPlanet	Wheter the setup contains a planet (0 or 1)	1
DiscIcPlanetRadius	Semi-major axis of the planet	1
DiscIcPlanetEccen	Eccentricity of the orbit	0
DiscIcPlanetIncl	Inclination of the orbit (in degrees)	0
DiscIcPlanetMass	Mass of the planet	1e-3
DiscIcPlanetAccretionRadiusHill	Accretion radius of the planet (in units of the Hill radius)	0.4
NDust	Number of dust particles	0
DustGasRatio	Dust to gas ratio when using dust	0.01

Table 1: Summary of the parameters controlling the disc setup and their default values

7.3 Load from external file

GANDALF can load two main file formats; simple ASCII column format and SEREN format. These are described in detail in Section 6. In order to read from file, you must set the following parameters :

- `ic = file` : Tell GANDALF to ignore initial conditions generators and read from file instead
- `in_file = ..` : The name of the initial conditions file
- `in_file_form = column/sf/su` : The format of the initial conditions file

7.4 Generate inside python script

GANDALF can generate initial conditions if using python scripts. The initial conditions can be first set in numpy arrays and then imported into the C++ code in order to perform the simulation. More information is given about this method, including detailed examples, in Section 8.3.

8 GANDALF Python library

The GANDALF Python library can be used to control a simulation via a Python script. In addition, it also provides routines for performing analysis tasks. Combined with the power of the Python language, this allows you to use it for many purposes. Some examples are listed below:

- Load in previously run simulations for analysis and producing visualisation
- Load in multiple previously run simulations for comparisons
- Prepare often-run simulations including the analysis in a single script
- Run batches of simulations (e.g. a parameter study) with a single controlling script
- Run or analyse simulations interactively through the interpreter
- Generate initial conditions directly in python (instead of via a file or in C++) and run the simulation

It also provides an easier entry into using SPH and N-body than other codes which require more investigation of the code mechanics and file formats before even basic simulations can be run.

There are two main ways to use the Python side of GANDALF; (a) an interpreter, which works similarly to a shell (e.g. bash, csh) (b) a python script. The interpreter understands a specific set of commands to load a simulation and plot quantities. You can use it for example to quickly read the output of a simulation and do a simple plot to check what is happening. Using a Python script instead is more powerful since, in addition to the same commands supported by the interpreter, you have the full power of Python at your disposal. For example you can access the raw particle data, compute additional particle properties, generate the initial conditions, and much more. The library will take care of the boring details such as reading the snapshots and extracting the data; you can concentrate on what to do with the data and on the science, and forget about the details.

The Python files for both the library and the interpreter are contained in the `gandalf/analysis` sub-directory. We explain briefly in the next sections how to run the interpreter or import the GANDALF library (if you are writing a script) and then an extensive tutorial follows which should clarify how to use the library.

8.1 Interpreter

The interpreter is located in the analysis/ folder in the main directory of GANDALF. To start the interpreter, type from the main GANDALF folder:

`python analysis/gandalf_interpreter.py`.

A list of commands available for the python interpreter can be printed by typing the 'help' command. Furthermore, typing 'help command' gives more information on the chosen command. The commands have a streamlined syntax so that you can perform the tasks you need with as little typing as possible. For example, to render the density of the sph particles in a 2d simulation, you can just issue the command `render x y rho`. Under the hood, this is translated in the python code `render('x','y','rho')`.

8.2 Python script - overview of the libraries to import

The GANDALF Python library contains a number of modules that can be imported in to provide the desired functionality.

- `gandalf.analysis.facade`
  This module is the main front-end to the GANDALF Python library and contains the Python-wrapped C++ executable and all functions required to set-up and run simulations. This module must always be imported in GANDALF Python scripts.

- `gandalf.analysis.compute`
  This module contains all extra and user-defined routines for computing important quantities from the simulation snapshot data. Currently contains routines for computing centre-of-mass properties, L1 error norm (when provided with an analytical solution) and the Lagrangian radius.
- `gandalf.analysis.data_fetcher`
  This module contains routines for exporting data from the C++ code to Python, generating custom data (e.g. time evolution) from snapshot data and generating user-defined data quantities from raw particle data.

8.3 GANDALF Python library tutorial

The GANDALF Python library contains a variety of commands that are used to create and run new simulations, load in old simulations from the disk, analyse and plot results and more. A full list of all of these commands is given in Appendix A. The same information can be found in the source code file `gandalf/analysis/facade.py`. Also, when running the interpreter, information on interactive commands can be obtained by typing `help` for a full list of available commands, or `help command` for information on that particular command. Finally, as common in Python, if running through a script one can get the documentation string of a function in the special variable called `__doc__`.

We provide here a short tutorial demonstrating from the most basic to the more advanced functionality of the Python library. These examples are also contained in the `gandalf/examples` sub-directory and should run as typed, although it is also useful for the user to write the examples themselves if unfamiliar with python syntax.

8.3.1 Example 1 - Creating and running a new simulation from a parameter file

This first example demonstrates how to set-up and run a new simulation from a given parameters file (nominally called `params.dat` in these examples).

```
#=======================================================================
# example01.py
# Basic example to run a simulation from a parameters file.
#=======================================================================
from gandalf.analysis.facade import *

# Create simulation object from parameters file
sim = newsim("adsod.dat")

# Perform all set-up routines and then run simulation to completion
setupsim()
run()
```

The first command

```
from gandalf.analysis.facade import *
```

loads in all definitions and functions from the GANDALF python frontend, `facade.py`. Note that this assumes that the `PYTHONPATH` environment variable has been correctly set-up. Otherwise, the full absolute path of the `facade.py` file must be given. The second command

```
sim = newsim('adsod.dat')
```

reads in the parameters file `'adsod.dat'` and creates a new simulation from those parameters and returns a Python object, `sim`, which can be used to refer to that simulation in Python (N.B. the user can use any name for this object, as long as it does not clash with another GANDALF or Python object). In this case, it is assumed the parameters file is selecting an initial conditions generator in the main code to generate the particles in the simulation. In order to fully set-up the simulation so it is ready to be run, the command

```
setupsim()
```

must be run. Finally, to run the simulation to completion, we must execute the command

```
run()
```

The code will run until the specified endtime and then exit the python environment.

If run through the interpreter, the same example would look like:

```
newsim adsod.dat
setupsim
run
```

8.3.2 Example 2 - Creating a simulation and modifying a parameter before running

This example demonstrates how to modify parameters in Python that have been loaded in from a given parameters file.

```
#=======================================================================
# example02.py
# Example to prepare a simulation from a parameters file, modify a parameter,
# then run the simulation to completion.
#=======================================================================
from gandalf.analysis.facade import *

# Create simulation from parameters file and modify a single parameter
sim = newsim("adsod.dat")
sim.SetParam("tend",1.5)

# Process modified parameters, set-up simulation and run to the end.
setupsim()
run()
```

The first two lines of this example are the as the first example, where we import the GANDALF Python frontend and then create a new simulation from the parameters file `'adsod.dat'`. Once loaded in, we can now modify parameters with the command

```
sim.SetParam(parameter_name,new_value)
```

In this case, we set the parameter `tend` (the total simulation run-time) to the value `2.0` with `sim.SetParam(`tend',2.0)`. Once all parameters have been modified, then we can set-up the simulation fully with `setupsim()` and run it with `run()`. Note that if we attempt to modify any parameters AFTER calling `setupsim()`, then the code will terminate with an exception. It is not possible to change the parameters of a simulation through the interpreter.

8.3.3 Example 3 - Generating initial conditions using internal routines without a parameters file

In this example, we will create and run a simulation without needing to load-in an external parameters file.

```
#=======================================================================
# example03.py
# Example to create a `blank' simulation object, set all important parameters,
# then run the simulation to completion.
#=======================================================================
from gandalf.analysis.facade import *

# Create `blank' simulation object (2-dimensional SPH)
sim = newsim(ndim=2,sim="sph")

# Set all important simulation parameters in order to create a grad-h SPH
# simulation of a 2D Sedov blast test initially from a 64x64 lattice.
sim.SetParam("ic","sedov")
sim.SetParam("run_id","SEDOV1")
sim.SetParam("Nlattice1[0]",64)
sim.SetParam("Nlattice1[1]",64)
sim.SetParam("boxmin[0]",-1.0)
sim.SetParam("boxmin[1]",-1.0)
sim.SetParam("boxmax[0]",1.0)
sim.SetParam("boxmax[1]",1.0)
sim.SetParam("boundary_lhs[0]","periodic")
sim.SetParam("boundary_rhs[0]","periodic")
sim.SetParam("boundary_lhs[1]","periodic")
sim.SetParam("boundary_rhs[1]","periodic")
sim.SetParam("dimensionless",1)
sim.SetParam("Nlevels",10)
sim.SetParam("tend",0.5)
sim.SetParam("tsnapfirst",0.0)
sim.SetParam("dt_snap",0.1)

# Now perform all set-up routines and run simulation
setupsim()
run()
```

In this example, we call the `newsim` function passing the dimensionality of the simulation instead of a parameter file. Therefore the line `sim = newsim(ndim=2,sim=`sph')` sets up a 2-dimensional SPH simulation, but is otherwise unspecified because no other parameters have been set. Next, we can set all the parameters that would have otherwise been set in the parameters file. In this example, we set up an SPH simulation with initial conditions to perform the Sedov blast-wave test (`sim.SetParam(`ic',`sedov')`); (See Section 5 for a full description of all parameters). Once all parameters have been set, we can fully set-up and run the simulation with `setupsim()` and `run()`.

8.3.4 Example 4 - Running a simulation and plotting results to screen and file

In this example, we show how to plot particle data interactively during a live simulation.

```
#=======================================================================
# example04.py
# Example to create a simple 2D random cube of particles and plot the x-y
# coordinates to screen as the simulation progresses.
```

```
#============================================================================
from gandalf.analysis.facade import *

# Create simulation object from `glass.dat' parameters file
sim = newsim("glass.dat")

# Set the time interval where the matplotlib window is updated in seconds
sim.SetParam("dt_python",2.0)

# Set-up simulation, plot the x-y coordinates and run the simulation
setupsim()
plot("x","y")
run()

# Finally, save the figure to file
savefig("figure.eps")
block()
```

In this simulation, we will run a live simulation while also plotting particle data in a matplotlib window as the simulation is still running. We first create the simulation as in previous examples with `newsim`. We then set the value of the `dt_python` which is the time between python/matplotlib window updates. After the `setupsim()` command, we can then plot the initial conditions from the simulation using the command

```
plot(x,y)
```

where `x` and `y` are the x- and y-axis plotting variables (N.B. these can be any variables, not necessarily x and y. e.g. `plot('y','z')`). If we now run the simulation with `run()`, the simulation will compute as normal while updating the plot window every `dt_python` seconds (in this casem every 2 seconds). After the simulation has finished, we can save the final plot to file with the command

```
savefig(filename)
```

where `filename` is the intended filename *including* the file extension. matplotlib automatically determines the correct file format from the extension (e.g. typing `savefig(`fig1.png')` will automatically save the window to a png file without need for additional arguments). The use of `block()` in the last line is needed because otherwise python would exit immediately from the script. Instead, the function pauses the script so that you can look at the plot. Pressing Enter (or any other key) causes to script to continue (and so in this case to terminate as there is no other instruction afterwards).

This same example can also be run through the interpreter (apart for changing the `dt_python` parameter). Note that there is no need to call block as the interpreter already pauses after each command in order to wait for the next one:

```
newsim glass.dat
setupsim
plot x y
run
savefig figure.eps
```

8.3.5 Example 5 - Reading a simulation from disc and plotting several snapshots

In this example, we load in a previously run simulation from the disk and generate various plots.

```
#============================================================================
```

```
# example05.py
# Example to load in a previously run simulation from the disk,
#=============================================================================
from gandalf.analysis.facade import *

# Load simulation with run_id `ADSOD1' (run in example 1)
loadsim("ADSOD1")

# Plot x vs. density for first snapshot and save to eps file
plot("x","rho",snap=0)
savefig("snap1.eps")

# Plot x vs. density for second snapshot and save to eps file
plot("x","rho",snap=1)
savefig("snap2.eps")

# Plot x vs. density for first and second snapshots and save to eps file
plot("x","rho",snap=0)
addplot("x","rho",snap=1)
savefig("snap12.eps")
block()
```

Instead of a creating a new simulation with the `newsim` function, we will load in a previous run simulation with the command

```
loadsim(runid)
```

where `runid` is the simulation run identification string. Note, this assumes that all files from that simulation are present in the folder, including the `runid.param` file which contains all the parameters used to run the simulation. In this case, we load a simulation with the runid `ADSOD1` with the command `loadsim(`ADSOD1')`. We can then plot particular snapshots of the simulation using the `plot` command with the optional `snap` argument. Therefore to plot the first snapshot, type `plot('x','y',snap=0)`, to plot the second snapshot, type `plot('x','y',snap=1)`, etc.. Note that Python uses C-style indexing, i.e. starting from zero, unlike other conventions like in FORTRAN where indexing starts from one. If we wish to overplot one snapshot over another, we first plot the first snapshot and then use the command

```
addplot('x','y',snap=1)
```

to add the second snapshot plot over the first one. This behaviour can also be replicated by adding the `overplot` optional boolean argument to `plot`, i.e. `plot(`x',`y',snap=1,overplot=True)`.

It is possible to run this example too through the interpreter:

```
loadsim ADSOD1
plot x rho snap=0
savefig snap1.eps
plot x rho snap=1
savefig snap2.eps
plot x rho snap=0
addplot x rho snap=1
savefig snap12.eps
```

From now on, we stop providing also the code for the interpreter. As you have seen, the conversion is trivial to do.

8.3.6 Example 6 - Reading and plotting multiple simulations

In this example, we show how to read in multiple simulations to make comparison plots.

```
#=======================================================================
# example06.py
# Load in an old simulation from the disk, run a new simulation and then
# plot both simulations in the same plot for comparison.
#=======================================================================
from gandalf.analysis.facade import *

# Load in Sod test simulation (run in example 1)
sim0 = loadsim("ADSOD1")

# Create new simulation object from `adsod.dat' parameters file, but modify
# to use no artificial viscosity and then run to completion.
sim1 = newsim("adsod.dat")
sim1.SetParam("run_id","ADSOD2")
sim1.SetParam("avisc","none")
setupsim()
run()

# Plot Sod test results, with (sim=0) and without (sim=1) artificial viscosity,
# on same figure and then save to eps file.
plot("x","rho",sim=0,snap=2)
addplot("x","rho",sim=1,snap=2)
savefig("sod12.eps")
block()
```

Multiple simulations can be loaded into GANDALF using the `loadsim` command. However, they must be returned to objects with different names; in this case, we name them `sim0` and `sim1` (to follow the C-style numbering convention). A snapshot from a particular simulation can be plotted by adding the optional argument `sim` to the plot command, i.e. `plot(`x',`rho',sim=0,snap=0)` to plot the first snapshot in the first simulation. The second simulation is then over-plotted with the command `addplot(`x',`rho',sim=1,snap=0)`. In this example, we load in an old simulation (as run in example 1) and then run a new simulation (the same initial conditions but without any artificial viscosity) and then plot the results on the same window and to an eps file.

8.3.7 Example 7 - Overplotting the analytical solution with the simulation results

In this example, we demonstrate how to overplot analytical solutions over particle plots.

```
#=======================================================================
# example07.py
# Overplot the analytical solution of a known problem while simulatneously
# running the simulation.
#=======================================================================
from gandalf.analysis.facade import *

# Create the new simulation object
sim = newsim("adsod.dat")
sim.SetParam("dt_python",2.0)
setupsim()

# Plot both the simulation results along with the analytical solution
```

```
# (N.B. the analytical solution is overplotted by default).
plot("x","rho")
plotanalytical("x","rho")

# Set the limits of the x-axis plot
limit("x",-1.1,1.1)

# Finally run the simulation to completion
run()
block()
```

For some sets of initial conditions (e.g. shocktubes), the analytical solution is provided in one of the python modules (`gandalf.analysis.analytical`). The python module will read in the values specified in the simulation parameters file and return the correct solution for that given snapshot time. To overplot a specified quantity (e.g. `x` vs density), then use the command

```
plotanalytical('x','rho')
```

Another useful command to set the limits of any plot quantity is

```
limit(quantity,min,max)
```

where `quantity` is a string of the quantity and `min` and `max` define the range of the plot. In this example, `limit(`x',-1.1,1.1)`

8.3.8 Example 8 - Creating initial conditions directly in the python script

In this example, we demonstrate how to generate some simple initial conditions inside a python script. For simplicity, we start with a simple 1D shocktube example.

```
#=======================================================================
# example08.py
# Create initial conditions for SPH simulation inside the python script, and
# then run the simulation to completion while plotting results.
#=======================================================================
from gandalf.analysis.facade import *
import numpy as np
import time

# Set basic parameters for generating initial conditions
Nhydro = 200
vfluid = 4.0
xmin = -1.5
xmax = 1.5

# Set uniform line of Nhydro particles between the limits of xmin and xmax
# in local numpy arrays
deltax = (xmax - xmin) / Nhydro
x = np.linspace(xmin + 0.5*deltax,xmax - 0.5*deltax,num=Nhydro)
m = np.ones(Nhydro)*(xmax - xmin)/Nhydro

# Set velocities of shock-tube so v = vfluid for x < 0 and -vfluid for x > 0
vx = np.ones(Nhydro)*vfluid
vx[x > 0.0] = -vfluid
```

```
# Create new 1D simulation object and set parameters
sim = newsim(ndim=1,sim='sph')
sim.SetParam('ic','python')
sim.SetParam('gas_eos','isothermal')
sim.SetParam('Nhydro',Nhydro)
sim.SetParam('tend',0.2)
sim.SetParam('dt_snap',0.05)
sim.SetParam('dimensionless',1)
sim.SetParam('vfluid1[0]',vfluid)
sim.SetParam('vfluid2[0]',-vfluid)
sim.SetParam('boxmin[0]',xmin)
sim.SetParam('boxmax[0]',xmax)
sim.SetParam('run_id','SHOCKTUBE1')

# Call setup routines and import particle data
sim.PreSetupForPython()
sim.ImportArray(x,'x')
sim.ImportArray(vx,'vx')
sim.ImportArray(m,'m')
setupsim()
# Plot the density of all particles near the shock
plot("x","rho")
plotanalytical("x","rho",ic="shocktube")
limit("x",-0.17,0.17,window="all")
limit("rho",0,21.0,window="all")

# Run simulation and save plot to file
run()
savefig("shocktube.png")
block()
```

As well as importing the GANDALF Python library, we must also import the NUMPY library, which is a popular mathematical library in Python for handling arrays, with the command

```
import numpy as np
```

We first create the initial conditions in Python before exporting the data to the C++ part of the code. This can be done by creating NUMPY arrays for each variable, e.g. `x`, `m`, etc.. Note that even for vector quantities (e.g. position, velocity), we must allocate separate 1D NUMPY arrays for each component, for example the commands

```
a1 = np.zeros(N),
a2 = np.ones(N),
a3 = np.linspace(0.0,1.0,num=N)
```

return NUMPY arrays of size `N` with all zeros, all ones, and uniform values between 0 and 1 respectively (See NUMPY website and documentation for more information on NUMPY functions). For this simple shocktube test, we set all velocities left of the shock interface to 4.0 and right of the shock interface to −4.0 with the commands `vx = np.ones(Nsph)*4.0; vx[x > 0.0] = -4.0`.

We then create an undefined 1D simulation object with `sim = newsim(ndim=1)` and set various parameters determing the simulation type, integration scheme, equation of state, etc.. We must then inform the code that we intend to generate initial conditions in the python script (and not say from an external file or one of the internal C++ routines) by the command `sim.SetParam('ic','python')`. We must also inform the code of the number of SPH particles to be used with the command `sim.SetParam('Nsph',Nsph)`, in order to allocate

enough memory for the simulation (N.B. `Nsph` is simply a local Python variable in this example. This can be any other variable, or even simply an integer number).

The last step before importing the arrays to C++, we must initialise various things in the code, including importantly allocating memory for the particles, by calling the function

```
sim.PreSetupforPython()
```

Next, we can import the array into the C++ arrays by the command

```
sim.ImportArray(numpyarray,varname)
```

where `numpyarray` is the local NUMPY array and `varname` is a string containing the C++ variable name. For example, to import the x-positions, we call the command `sim.ImportArray(x,'x')`. Once all arrays have been called, we can finally call the `setupsim` function and plot and run the simulation to completion.

8.3.9 Example 9 - Creating initial conditions for N-body simulation in python script

In this example, we demonstrate how to generate some simple initial conditions for pure N-body simulation inside a python script.

```
#=======================================================================
# example09.py
# Create initial conditions for pure N-body simulation inside the python
# script, and then run the simulation to completion while plotting results.
#=======================================================================
from gandalf.analysis.facade import *
import numpy as np
import time

# Create empty numpy arrays for setting star initial conditions
Nstar = 3
x = np.zeros(Nstar)
y = np.zeros(Nstar)
vx = np.zeros(Nstar)
vy = np.zeros(Nstar)
m = np.zeros(Nstar)
h = 0.000001*np.ones(Nstar)

# Set values for each star individually (Note all velocities initially zero)
m[0] = 3.0;   x[0] = 1.0;   y[0] = 3.0
m[1] = 4.0;   x[1] = -2.0;  y[1] = -1.0
m[2] = 5.0;   x[2] = 1.0;   y[2] = -1.0

# Create new 1D simulation object and set parameters
sim = newsim(ndim=2,sim='nbody')
sim.SetParam('ic','python')
sim.SetParam('nbody','hermite4ts')
sim.SetParam('sub_systems',0)
sim.SetParam('Npec',3)
sim.SetParam('Nlevels',1)
sim.SetParam('Nstar',Nstar)
sim.SetParam('tend',80.0)
sim.SetParam('dt_snap',1.0)
sim.SetParam('noutputstep',128)
```

```
sim.SetParam('ndiagstep',2048)
sim.SetParam('dimensionless',1)
sim.SetParam('run_id','BURRAU1')
sim.SetParam('out_file_form','su')

# Call setup routines and import particle data
sim.PreSetupForPython()
sim.ImportArray(x,'x','star')
sim.ImportArray(y,'y','star')
sim.ImportArray(vx,'vx','star')
sim.ImportArray(vy,'vy','star')
sim.ImportArray(m,'m','star')
sim.ImportArray(h,'h','star')
sim.SetupSimulation()

# Plot the density of all particles near the shock
plot("x","y",type="star")
limit("x",-30.0,30.0,window="all")
limit("y",-20.0,40.0,window="all")

# Run simulation and save plot to file
run()
block()
```

8.3.10 Example 10 - Generating rendered images from SPH simulations

In this example, we show how to generate rendered images of some quantity. Note that rendered plots can only be created for simulations using 2 or 3 dimensions.

```
#=======================================================================
# example10.py
# Create a rendered image during an interactive simulation.
#=======================================================================
from gandalf.analysis.facade import *

# Create simulation object from Kelvin-Helmholtz parameters file
sim = newsim("khi.dat")
setupsim()

# Generate rendered image of density field on the x-y plane with a
# resolution of 128 x 128 pixels.
render("x","y","rho",res=128)
limit("x",-0.5,0.5)
limit("y",-0.5,0.5)
limit("rho",1.0,2.0)
run()
block()
```

A rendered plot of the density on the x-y plane can be created using the command

```
render('x','y','rho')
```

The first argument specifies the x-coordinate, the second argument the y-coordinate and the third argument the rendered quantity. An important optional argument is `res` which specifies the resolution (in pixels) of

the rendered image in each dimension. If the image is not square, then the user may pass the resolution in each dimension in parenthesis, e.g. `render(`x',`y','rho',res=(128,64))`.

8.3.11 Example 11 - Plotting in alternative coordinate systems

In this example, we show how to plot in alternative coordinate systems (i.e. other than Cartesian coordinates).

```
#=======================================================================
# example11.py
# Plot particle quantities in an alternative coordinate system.
#=======================================================================
from gandalf.analysis.facade import *

# Create simulation object from Kelvin-Helmholtz parameters file
sim = loadsim("SEDOV1")

# Plot the density as a function of radial position
plot("R","rho",snap=5)
block()
```

Once the simulation has been loaded or set-up as usual, then it is possible to plot in 3 pre-defined coordinate systems, Cartesian, spherical polar and cylindrical polar coordinates, or even a mixture. Cartesian coordinates are represented by the strings, `x`, `y`, `z`; sphericial polar coordinates by the strings `r`, `theta`, `phi`; cylindrical coordinates by the strings `R`, `phi`, `z`. For example, to plot the density as a function of radius (for spherically symmetric simulations), then we can use the command

```
plot('r','rho')
```

8.3.12 Example 12 - Changing the plotting units

In this example, we show how to plot using different units to those used in the simulation or provided in the snapshot files.

```
#=======================================================================
# example12.py
# Plot particle quantities in an alternative coordinate system.
#=======================================================================
from gandalf.analysis.facade import *
from matplotlib.colors import LogNorm

# Create simulation object from Boss-Bodenheimer parameters file
sim = newsim("bossbodenheimer.dat")
sim.SetParam("tend",0.02)
setupsim()

# Run simulation and plot x-y positions of SPH particles in the default
# units specified in the `bossbodenheimer.dat' parameters file.

plot("x","y")
addplot("x","y",type="star")
limit("x",-0.007,0.007)
```

```
limit("y",-0.007,0.007)

window()
render("x","y","rho",res=256,#norm=LogNorm(),
       interpolation='bicubic')
limit("x",-0.007,0.007)
limit("y",-0.007,0.007)

run()
block()

# After pressing return, re-plot last snapshot but in new specified units (au).
window(1)
plot("x","y",xunit="au",yunit="au")
window(2)
render("x","y","rho",res=256,#norm=LogNorm(),
       interpolation='bicubic')
limit("x",-0.007,0.007)
limit("y",-0.007,0.007)
block()
```

In order to plot quantities using a different unit to the default, the optional arguments `xunit` and `yunit` must be appended to the argument list for the relevant plotting command. These arguments contain strings of the required unit (See section 5.2 for the list of available units).

8.3.13 Example 13 - Creating and plotting user-defined quantities

In this example, we show how to create new quantities to be plotted as regular quantities by the python library.

```
#===============================================================================
# example13.py
# Create a new user-defined quantity and plot on window.
#===============================================================================
from gandalf.analysis.facade import *

# Load in simulation frmo disk
sim = loadsim("ADSOD1")

# Create new quantity, specific kinetic energy of particles, including the
# scaling factor (specific energy) and latex label.
CreateUserQuantity("ke","0.5*vx*vx",scaling_factor="u",
                   label="$\\frac{1}{2}v^2$")

# Plot defined quantity along with internal energy
plot("x","ke",snap=2)
limit("ke",0.0,2.7)
addplot("x","u",snap=2)
block()
```

We can create new quantities from existing particle properties using the command

`CreateUserQuantity(quantity_name,quantity_formula)`

where `quantity_name` is the new string name of the derived quantity and `quantity_formula` is string

containing an algebraic formula that defines the new quantity. In our example, we create a new quantity to compute the kinetic energy of a particle with the algebraic formula `0.5*m*vx*vx`. For 2D and 3D simulations, this would become `0.5*m*(vx*vx + vy*vy)` and `0.5*m*(vx*vx + vy*vy + vz*vz)` respectively. Note that the algebraic formula can also use parameters by using the same string as defined in the parameters file.

The new quantity can then be plotted as any other regular quantity, e.g. `plot('x','ke')`. Two important optional arguments are `scaling_factor`, which defines the string of the unit (e.g. in this case `E` since `ke` is an energy), and `label`, which is a latex string used on the plots (e.g. `frac{1}{2}mv^2`).

8.3.14 Example 14 - Plotting time series of particle properties

In this example, we show how to plot a time series of particle quantities from each snapshot recorded in the simulation buffer. This can be used to plot useful quantities versus time (e.g. mass of a sink versus time), or x-y tracks of a particle as it moves in space.

```
#=======================================================================
# example14.py
# Plot quantities of single particles as a function of time through all
# snapshots in the simulation.
#=======================================================================
from gandalf.analysis.facade import *

# Load in simulation from disk (from example 12)
sim = loadsim("BB1")

# Plot the density of particle `100' as a function of time
time_plot("t","rho",id=100,linestyle="-")
block()

# Now plot x-y track of particle 100 as it moves across computational domain
time_plot("x","y",id=100,linestyle="-")
block()
```

In order to plot a time series of some particle property, we use the command

```
time_plot('t','rho',id=0)
```

where `id` is the unique/original id of the particle to be plotted. If no id is given, then an error is returned and nothing will be plotted on screen.

8.3.15 Example 15 - Creating an animation from simulation snapshot files

In this example, we show how to generate a movie from a series of snapshot files loaded into memory.

```
#=======================================================================
# example15.py
# Generate a movie from the snapshots of the simulation
#=======================================================================
from gandalf.analysis.facade import *

#Load in simulation from the disk (run in example01.py)
sim = loadsim('ADSOD1')
```

```
#Make a plot
plot('x','rho')

#Loop over all the snapshots and produce a movie
make_movie('MOVIE1.mp4')
block()
```

Before generating a movie, we must first specify the plot commands that are to be used to generate the movie, for example with the regular plot command `plot('x','rho')`. Once the commands to generate the plot window have been completed, we then generate the movie with the command

`make_movie('MOVIE1.mp4')`

where the only required argument is the filename of the movie. The python library will then step through each snapshot in turn, generate the image as a temporary png file, and then attach all images together using ffmpeg. If you do not have ffmpeg installed, the command will fail. All temporary png files should then be deleted automatically from the disc.

8.3.16 Example 16 - Retrieving data from the simulation

In this example, we show how to save some data from the snapshots in a variable. You can even save the rendered image.

```
#=============================================================================
# example16.py
# Retrieve data and save it inside a variable
#=============================================================================
from gandalf.analysis.facade import *

# Load in simulation from disk (from example10.py)
sim = loadsim("KHI1")

# Get the data in variable x
x=get_data('x')
print x

# Do a rendered plot and save the image
image=get_render_data('x','y','rho')

# Use the matplotlib to plot the image
import matplotlib.pyplot as plt
plt.imshow(image,interpolation='nearest')
plt.show()

# You can also the do the following when doing a plot:
data=plot('x','y')

# Let's do a matplotlib plot in a new figure to show how you can access the data
plt.figure()
plt.plot(data.x_data,data.y_data,'.')

# It works with render too
plt.figure()
data=render('x','y','rho')
```

```
plt.figure()
plt.imshow(data.render_data,interpolation='nearest')
plt.show()
```

The syntax of the functions get_data and get_render_data should be self-explanatory. These routines just return the requested quantity as a numpy array, that you can save in a variable for further processing. Alternatively, every time you do a plot you can also save the return value in a variable as shown further on in the example. Particle plots save the data used on the x-axis inside the `x_data` field and on the y-axis inside the `y_data` field. Render instead saves the image in the `render_data` field. In this way, you can use the rendered image to grid your SPH data. At the moment render only renders to a 2d grid; however, we plan in the future to extend its capabilities to render to a 3d grid.

8.3.17 Example 17 - Creating and plotting user-defined quantities from a function given by the user

In this example, we show how to create and plot an user-defined quantity. However, rather than building it from a mathematical formula as shown in the example 13, we compute it using a function provided by the user. This allows you to have much more flexibility in performing the computations you want with the data of the simulation.

```
#===============================================================================
# example17.py
# Create a new user-defined quantity from a function and plot on window.
# Extends on example13.py
#===============================================================================
from gandalf.analysis.facade import *

# Load in simulation frmo disk
sim = loadsim("ADSOD1")

# Create new quantity, specific kinetic energy of particles, including the
# scaling factor (specific energy) and latex label.
CreateUserQuantity("ke","0.5*vx*vx",scaling_factor="u",
                   label="$\\frac{1}{2}v^2$")

# Plot defined quantity along with internal energy
plot("x","ke",snap=2)
limit("ke",0.0,2.7)
addplot("x","u",snap=2)
block()

# Define a function for computing kinetic energy
def ComputeKineticEnergy(snap,type="default",unit="default"):
    vx=get_data("vx",snap=snap,type=type,unit=unit)
    return 0.5*vx*vx

# Create another new quantity
CreateUserQuantity("ke2",ComputeKineticEnergy,scaling_factor="u",
                   label="$\\frac{1}{2}v^2$")
# Overplot the new quantity - see that it overlays perfectly on the previous one
addplot("x","ke2",snap=2)
block()
```

Refer to the example 13 for the first use of `CreateUserQuantity`. For what concerns the second, we first need to define the function that we want to use for computing the data. In this case, the function just retrieves the array

with the velocities (see the previous example) and computes the kinetic energy from them. The function needs to return the computed value. Note that we pass the parameters specifying the snapshot, the type of particle (e.g., gas or star) and the unit to `get_data`. This is very important to do; if you forget to do it, you might get data from the wrong snapshot, the wrong particle type, or in the wrong units.

8.3.18 Example 18 - Creating and plotting time series from a function given by the user

In this example, we show how to plot time series from functions defined by the user.

```
#=============================================================================
# example18.py
# Plot quantities as a function of time through all snapshots in the simulation.
# Extends on example14.py
#=============================================================================
from gandalf.analysis.facade import *
#In this case, we need also a function defined in compute
from gandalf.analysis.compute import lagrangian_radii

# Load in simulation from disk (from example 12)
sim = loadsim("BB1")

# Define the half-mass radius
CreateTimeData("half_r",lagrangian_radii,mfrac=0.5)

# Plot it
time_plot("t","half_r")
block()

# Define a function for computing the total mass
def ComputeMass(snap,type="default",unit="default"):
    m=get_data('m',snap=snap,type=type,unit=unit)
    return m.sum()

# Define the quantity
CreateTimeData("totm",ComputeMass)

# Plot it
time_plot("t","totm")
block()
```

For accomplishing this task we need to use the function `CreateTimeData`. Note that the way this function works is very similar to `CreateUserQuantity` shown in the previous example. In the first example we use the function `lagrangian_radii`, which is defined in GANDALF in the module `compute`. Note that we can pass additional parameters to the function using keywords; for example in this case we specify that we want the half-mass radius using the keyword `mfrac`. In the second example, we show how to define your own function for computing a quantity from a snapshot. Just remember that in this case (differently from the previous example) you need to return a scalar, which will be plotted as a function of time. The example retrieves the array with the masses (see example 16) and sums up all of them to get the total one.

8.4 Tips and tricks

To plot the results we use the well-known library matplotlib. This means that if you are running from a script you have several ways to customize the plots if you are not satisfied by their graphical appearance:

- Calling functions in matplotlib. Example: to logscale the y axis, just call `plt.yscale('log')` (assuming you have imported matplotlib).
- Passing keyword arguments to the plotting functions defined by GANDALF. Example: if you are doing a particle plot and want to change the marker size, you can call `plot('x','y',ms=2)`, and the ms keyword will be passed to matplotlib.
- Modifying the defaults for matplotlib using the .matplotlibrc files (see matplotlib documentation at http://matplotlib.org/users/customizing.html).

You might notice that the matplotlib window freezes when you are running a simulation live. This is not GANDALF's fault unfortunately, but a flaw in the design of the Python interpreter (if you want to know more, read here: https://wiki.python.org/moin/GlobalInterpreterLock). If you do require interaction with the matplotlib window (for example to be able to pan/zoom while the simulation is running), you can run the plots in another process. In order to do this, open the file defaults.py in the analysis folder and change the value of `parallel` to `True`. There is a drawback in this case: given that the plots are on another process, you cannot call matplotlib functions to customise your plots (but the other two ways we suggest for customising the plots still work).

9 Dust Dynamics

9.1 Theory

The equations of motion for the mutual dynamics of gas and dust coupled by drag forces are

$$\rho_g \frac{\mathrm{D}_g \mathbf{v}_g}{\mathrm{D}t} = -\nabla P + \rho_g \mathbf{a}_{g,\mathrm{ext}} + K(\mathbf{v}_d - \mathbf{v}_g), \tag{1}$$

$$\rho_d \frac{\mathrm{D}_d \mathbf{v}_d}{\mathrm{D}t} = \rho_d \mathbf{a}_{d,\mathrm{ext}} - K(\mathbf{v}_d - \mathbf{v}_g), \tag{2}$$

where $\frac{\mathrm{D}_g}{\mathrm{D}t}$ is the Lagrangian derivative with respect to the gas. The external accelerations are included to take into account additional forces, which may either be the same for both the gas and dust particles (such as gravity) or affect just one species (e.g. viscosity). These equations can be re-written in terms of an equation of motion for the relative and barocentric velocities, $\mathbf{v} = (\rho_g \mathbf{v}_g + \rho_d \mathbf{v}_d)/(\rho_g + \rho_d)$ and $\Delta\mathbf{v} = \mathbf{v}_d - \mathbf{v}_g$, (Youdin & Goodmann, 2005; Laibe & Price, 2014; Lorén-Aguilar & Bate, 2015),

$$\frac{\mathrm{D}\mathbf{v}}{\mathrm{D}t} = \frac{-\nabla P + \rho_g \mathbf{a}_{g,\mathrm{ext}} + \rho_d \mathbf{a}_{d,\mathrm{ext}}}{\rho_g + \rho_d} - \mathbf{F}, \tag{3}$$

$$\frac{\mathrm{D}\Delta\mathbf{v}}{\mathrm{D}t} = -\frac{\Delta\mathbf{v}}{t_s} + \mathbf{a}_{d,\mathrm{ext}} + \frac{\nabla P}{\rho_g} - \mathbf{a}_{g,\mathrm{ext}} - (\Delta\mathbf{v}\cdot\nabla)\mathbf{v} - \mathbf{G}, \tag{4}$$

where $t_s = \rho_g \rho_d / K(\rho_g + \rho_d)$. The functions $\mathbf{F}$ and $\mathbf{G}$ are

$$\mathbf{F} = \frac{1}{\rho_d + \rho_g} \nabla \cdot \left(\frac{\rho_d \rho_d}{\rho_d + \rho_d} \Delta\mathbf{v}\Delta\mathbf{v} \right), \tag{5}$$

$$\mathbf{G} = \frac{1}{\rho_d + \rho_g} \left[\rho_g \Delta\mathbf{v} \cdot \nabla \left(\frac{\rho_g}{\rho_g + \rho_d} \Delta\mathbf{v} \right) - \rho_d \Delta\mathbf{v} \cdot \nabla \left(\frac{\rho_d}{\rho_g + \rho_d} \Delta\mathbf{v} \right) \right]. \tag{6}$$

These two formulations naturally lead to two different numerical approaches (in the same spirit as Eulerian vs. Lagrangian hydro codes), with equations (1) and (2) naturally represented by two different particle types representing fluid (gas) particles and dust particles. Equations (3) – (6) are more naturally represented as a single fluid that moves with the barycentric velocity, on top of which the dust fraction is advected relative to the barycentre at velocity $\Delta\mathbf{v}$ (see Laibe & Price 2014 for more details).

In GANDALF we solve directly equations (1) and (2) using a multi-fluid approach, using the semi-implicit time-stepping following Booth, Sijacki & Clarke (2015) and Lorén-Aguilar & Bate (2015). In these approaches we use of $\langle\Delta\mathbf{v}\rangle$, the average of the relative velocity in the time-step $[t, t + \Delta t]$. This is found by solving equation (4) under the approximation that the densities, accelerations and t_s are constant in both time and space. We find

$$\langle\Delta\mathbf{v}\rangle \frac{\Delta t}{t_s} = \Delta\mathbf{v}(t + \Delta t)\left(1 - \exp(-\Delta t/t_s)\right) - \left(\mathbf{a}_{d,\mathrm{ext}} + \frac{\nabla P}{\rho_g} - \mathbf{a}_{g,\mathrm{ext}} \right) \left[(\Delta t + t_s)\left(1 - \exp(-\Delta t/t_s)\right) - \Delta t \right], \tag{7}$$

where $\Delta\mathbf{v}(t + \Delta t) = \Delta\mathbf{v}(t) + \left(\mathbf{a}_{d,\mathrm{ext}} + \nabla P/\rho_g - \mathbf{a}_{g,\mathrm{ext}} \right) \Delta t$. From equation (7) the gas and dust accelerations can be found to be

$$\mathbf{a}_{g,\mathrm{drag}} = +\frac{\rho_d}{\rho_d + \rho_g} \frac{\langle\Delta\mathbf{v}\rangle}{t_s}, \tag{8}$$

$$\mathbf{a}_{d,\mathrm{drag}} = -\frac{\rho_g}{\rho_d + \rho_g} \frac{\langle\Delta\mathbf{v}\rangle}{t_s}. \tag{9}$$

When the approximations made above hold exactly the resulting accelerations are exact since equations (1) and (2) are linear as long as the drag coefficient does not depend on Δv. The change in internal energy is computed directly from the change in kinetic energy to ensure conservation.

9.2 Implementations

GANDALF now includes two implementations of combined dust-gas dynamics, based upon equations (1) & (2) and (7) – (9). Currently these are only implemented for the grad-h SPH algorithm. In both implementations the dust and gas are represented by separate particle types (which are identified in the code via the `ptype` particle attribute). In the first implementation the back-reaction of the dust on the gas is neglected (equivalent to $\rho_d \rightarrow 0$) – the gas dynamics are not modified when dust is included (test-particle limit). While in the second implementation the back-reaction on the gas is included (full two-fluid). The advantages and disadvantages of the methods are discussed below. These implementations can be found in src/Common/Dust.cpp

9.2.1 Test-particle implementation

The test-particle implementation closely follows Booth et al (2015). Since in general the gas particles and dust particles will not be at the same locations, the gas properties needed for evaluating equation (7) must be interpolated to the location of the dust particle. This is done in the standard SPH way. For each dust particle we compute a 'gas smoothing length' set by the local number density of gas particles, in a similar way to that of the gas. The gas properties are then interpolated to the location of the dust particles using a the standard kernel sum and the interpolated values are used in evaluating the drag acceleration.

9.2.2 Full two-fluid algorithm

The full two-fluid algorithm follows the improved semi-implicit algorithm of Lorén-Aguilar & Bate (2015). In order to ensure conservation of both energy and angular momentum the forces must: (1) change sign under the exchange of particles, (2) be directed along the lines between interacting particles (Laibe & Price 2012). (1) can be achieved simply by interpolation, but (2) requires forces to be directed between particles rather than in the direction of the velocity difference. Following Laibe & Price (2012) and Lorén-Aguilar & Bate (2015) we project the drag force between pairs of particles and evaluate the total acceleration using a kernel sum:

$$\mathbf{a}_{d,\mathrm{drag}} = -\sum_{i}^{\mathrm{gas}} \frac{m_i}{\rho_{g,i}} (\mathbf{S}_{id} \cdot \hat{\mathbf{r}}_{id}) \hat{\mathbf{r}}_{id} D(\mathbf{r}_{id}, h_i)$$

$$\mathbf{a}_{g,\mathrm{drag}} = -\sum_{i}^{\mathrm{dust}} \frac{m_i}{\rho_{d,i}} (\mathbf{S}_{ig} \cdot \hat{\mathbf{r}}_{ig}) \hat{\mathbf{r}}_{ig} D(\mathbf{r}_{ig}, h_g),$$

where $\hat{\mathbf{r}}_{ij} = (\mathbf{r}_i - \mathbf{r}_j)/|\mathbf{r}_i - \mathbf{r}_j|$ and

$$\mathbf{S}_{ij} = \frac{\rho_i}{\rho_i + \rho_j} \frac{\langle \Delta \mathbf{v}_{ij} \rangle}{t_s}. \tag{10}$$

For the kernel D we use the double hump version of the kernel used for the SPH forces $D(r,h) \propto (r/h)^2 W(r,h)$, which reduces the bias in the direction of the force estimate from nearby neighbours when compared with bell-shaped kernels (Laibe & Price, 2012).

9.2.3 Advantages, limitations and reasons for caution

- *Conservation* – The full fluid has the advantage that the total energy, momentum and angular momentum are conserved explicitly, while in the test-particle limit these are conserved for the gas particles, but not for the dust. In some problems, such as when radial drift is important the explicit conservation can help maintain the correct solution.
- *Dissipation* – The full two-fluid system is a dissipative system, with the drag forces converting kinetic energy into thermal energy. If the smoothing length, $h < c_s t_s$, this can lead to massive numerical dissipation especially when $\rho_d \approx \rho_g$. The semi-implicit algorithms help to reduce the dissipation (see Lorén-Aguilar & Bate, 2014), but this can still be an issue. For $\rho_d \ll \rho_g$ the dissipation is weak, with no dissipation occurring in the test particle limit.

- *Force Errors* – The standard SPH interpolation used in the test-particle implementation is generally accurate to ~ 0.1 per cent for the cubic spline, and even more accurate for smoother kernels (e.g. the Quintic spline). However, the projected interpolation is considerably more sensitive to the particle distribution. The double hump kernels help reduce the errors due to the particle distribution, but even for regular particle distributions the error can be ~ 1 per cent. When strong density gradients are present (e.g. the vertical structure of discs), the errors can be much larger (c.f the settling test, Lorén-Aguilar & Bate, 2014). For this reason smoother kernels are highly recommended when using the full-two fluid implementation.
- *Individual particle time-steps* – Individual particle time-steps are implemented for both dust algorithms, but typically the full-two fluid algorithm should not be used with individual particle time-steps, since the time averaged drag force is only correct once the time-step has been completed. Order unity errors can arise even the simplest dustybox tests when the time-step for the gas and dust differ for this reason, making it clear the exact conservation is necessary for accurate simulations with this algorithm. Development of a robust multi-step scheme for the full two-fluid scheme is being considered. Conversely, individual particle time-steps have been used in Booth, Sijacki & Clarke (2015) and Booth & Clarke (2016) with the test-particle limit algorithm and prove to be robust because the predicted velocities are not needed for force calculations. Individual particle time-steps have not been tested extensively in the GANDALF test-particle implementation, but preliminary tests have been successful.
- *Sinks* – The sink routines have not been modified to support dust, but in principal should work. However, I would recommend checking this when both dust and sink particles are needed in the same simulation. Accretion using the 'vacuum cleaner' type sinks *should* work properly, but has not been rigourously tested. The smoothed accretion mode is more likely to give issues since it removes mass from the particle closest to the sink only so it is possible that mass would end up being removed from the gas or dust particles only.

9.3 Drag Laws

Several drag laws have been implemented in via functions form of $t_s \equiv t_s(\rho_g, \rho_d, c_s)$. The different functional forms currently available can be chosen via the parameter options and are described below and are defined in src/-Headers/DragLaws.h. Most of these drag laws are phenomelogical in form, and only work with dimensionionless form. The one exception is the Epstein drag law, in which the stopping time is given by:

$$t_s = \frac{3}{4}\sqrt{\frac{\pi}{8}}\frac{m}{A(\rho_g + \rho_d)c_s}, \tag{11}$$

where m is the grain mass and A is its grain area (πa^2 for spherical grains). In the code, absorb the area-to-mass ratio into a single coefficient.

9.4 IO and initial conditions

Dust particles are currently supported only in the SEREN formats (both formatted and unformatted), with the dust particles following the gas particles on the disc. The number of dust particles is stored in the header, as described in section 6.2. The column format is *not* formally supported, but if used it will currently still contain the dust particles after the gas particles as long as MPI is not used. However, the number of dust particles is not currently saved in the column format.

Currently the only systematic way to generate initial conditions that include dust is to provide an IC file in the SEREN format that includes dust. For the test problems currently included, when dust is turned on (via setting the drag_forces parameter) the initial conditions generated are identical to those for the gas, although dust-to-gas ratio can be varied. Currently, generating multi-species initial conditions in python is also not supported (with the exception of star particles), although we hope to support this soon.

9.5 Test Problems

Some simple two-fluid test problems have been included in the directory tests/dust tests. Most of these are straight-forward modifications of classic hydrodynamics test problems to include dust particles. A description of a number of these can be found in Laibe & Price (2012 a,b), including the DUSTYBOX, DUSTYWAVE, DUSTYSHOCK and DUSTYSEDOV problems. The dust to gas ratio can be verified, but otherwise the dust initial conditions are identical to the those in the gas in all the problems provided.

9.6 Dust parameter options

- `drag_forces` : Select the drag force algorithm

none	= No drag forces
test particle	= Test particle limit drag regime
full twofluid	= Full two fluid drag regime

- `drag_law` : Select the drag law

fixed	$t_s = 1/K_D$
density	$t_s = 1/(K_D(\rho_d + \rho_g))$
epstein	$t_s = \frac{(3/4)\sqrt{\pi/8}}{K_D(\rho_d+\rho_g)c_s}$, where K_D is the area-to-mass ratio
LB2012	$t_s = \rho_d\rho_g/(K_D(\rho_d + \rho_g))$

- `drag_coeff` : Sets K_D
- `dust_mass_factor` : Sets the dust-to-gas mass ratio for the test problems

Including additional drag forces is straight forward. First add a new class to src/Headers/DragLaws.h. The new drag law can then be added to DustFactory::ProcessParameters (at the bottom of src/Common/Dust.cpp), which is responsible for selecting the desired drag law.

10 Developer notes

GANDALF makes heavy use of the object-oriented features in C++ in order to write a flexible code that can used in many different ways while maintaining good performance. For developers that wish to modify sections of GANDALF, or add new classes or functions, it is recommended to read this section in order to understand some of the ...

10.1 Class hiearchy

GANDALF uses a hierarchy of classes using inheritance to structure the code in a logical and maintainable manner. Although classes permeate the code at almost all levels, we briefly discuss the main classes here.

10.1.1 Simulation class

The `Simulation` class is the class that contains every other class, including SPH and N-body classes, in order to fully run the simulation. The `Simulation` class is split into three levels :

- `SimulationBase` - Non-templated base class for generating the simulation class. Provides binding layer between C++ code and Python (since Python cannot link to templated classes). Contains all subroutines that can be called by Python.
- `Simulation` - Main parent simulation class that contains all common functions for all possible simulations. Inherits from `SimulationBase`.
- `AlgorithmSimulation` - Child class containing specific algorithmic implementation. For example, 'grad-h' SPH simulations are run by the `GradhSphSimulation` class; pure N-body simulations are run by the `NbodySimulation` class, etc..

10.1.2 Hydrodynamics class

10.1.3 SmoothingKernel class

10.1.4 NeighbourSearch class

10.1.5 EOS class

10.1.6 EnergyEquation class

10.1.7 Nbody class

10.1.8 Sinks class

10.2 Templates

Templates are used in C++ to write generic code that can be used by different data types throughout the code without needing to write multiple versions of essentially the same code. It can also be used as an optimisation technique for performance, which we explain later.

The simplest case of using templates in GANDALF is for dimensionality. For almost all of the code, we use the template parameter '`ndim`' to generate a version of the code for each dimensionality used, i.e. 1, 2 and 3. In principle, `ndim` could simply be a variable which is defined by the user depending on which dimensionality is used for that simulation. However, the value of `ndim` would then need to be accessed in memory very frequently. By defining `ndim` as a template parameter, `ndim` will be defined statically and therefore does not require the overhead of accessing the memory address each time.

Throughout the code, the variables that are used as template parameters are :

- `ndim` : Number of dimensions
- `Kernel` : Smoothing kernel function used
- `ParticleType` : Particle data structure for employed hydrodynamics method
- `CellType` : Radiation transport tree cell-type, depending on employed RT method

10.3 Particle data structures

To optimise the speed of the code, GANDALF uses data structures that contain only the quantities for that particular hydrodynamics algorithm. For example, 'grad-h' SPH in 1D requires a very different number of variables to Godunov SPH in 3D. Therefore, a different version of each sub-routine that uses the SPH particle data directly is required using templates.

The various particle data structures are defined in the header file `'SphParticle.h'`. All particles are derived from a base data-type called `'Particle`

10.4 Particle array pointers

If a class which does not know the exact particle data structure wishes to access an array, or a single array element.

11 Units and scaling

In GANDALF, all physical calculations are done in dimensionless units for precision and accuracy reasons. However, the scaling factors for converting the initial conditions and parameters (in physical units) are calculated automatically, once the user has specified the units of the quantities in the parameters file.

11.1 Calculating scaling factors

Most physical quantities in GANDALF are some combination of the three basic physical unit types, length, mass and time. In principle, we are free to choose any scaling factor in order to convert to dimensionless units

$$r' = \frac{r}{R_0} \qquad m' = \frac{m}{M_0} \qquad t' = \frac{t}{T_0} \tag{12}$$

where r, m, t are the physical quantities, r', m', t' are the dimensionless analogues and R_0, M_0, T_0 are the scaling factors. The internal scaling factors for most quantities is some combination of the scaling factors of these three basic quantities. For example, the scaling factor for the dimensionless velocity, $v' = v/V_0$ is equal to $V_0 = R_0/T_0$.

11.2 Scaling factors for `G=1`

If we take some equation using `G` (e.g. Newton's law of gravity) and then substitute in the scaling factors and rearrange, we obtain :

$$a = \frac{G\,m}{r^2} \quad \Rightarrow \quad \frac{R_0}{T_0^2}\,a' = \frac{G\,M_0\,m'}{r'^2\,R_0^2} \quad \Rightarrow \quad a' = \underbrace{\left\{\frac{G\,M_0\,T_0^2}{R_0^3}\right\}}_{G'}\,\frac{m'}{r'^2}\,. \tag{13}$$

The final dimensionless equation has the same form as the original equation, with all constants and scaling factors grouped together in the 'dimensionless gravitational constant', `G'`. In N-body codes, it is common to adjust one of the scaling factors in order to set this new constant equal to unity. By convention, it is the time scaling factor that is adjusted :

$$\frac{G\,M_0\,T_0^2}{R_0^3} = 1 \quad \Rightarrow \quad T_0 = \left(\frac{R_0^3}{G\,M_0}\right)^{1/2} \tag{14}$$

For example, typical N-body units (at least in star cluser/formation simulations) of $R_0 = 1\,\mathrm{pc}$ and $M_0 = 1\,\mathrm{M}_\odot$ give a time scaling factor of $T_0 = 14.91\,\mathrm{Myr}$. It is important to realise that this then has a knock-on effect on any other physical quantity that contains the dimensions of time. For example, the unit of velocity becomes $V_0 = R_0/T_0 = 0.065\,\mathrm{km\,s^{-1}}$.

11.3 Temperature scaling factor

In thermodynamics, one additional unit that is often used (more often in the output rather than in internal quantities) is the thermodynamic temperature. This quantity represents an additional unit that cannot be represented in any combination of length, mass and time. The conversion between the three principle quantities is achieved via the Boltzmann constant. Just as with setting $G = 1$, we can also combine and convert quantities involving temperature to an appropriate dimensionless unit, $T' = T/\theta_0$, by effectively setting the quantity k_b/m_h equal to unity, i.e.

$$c^2 = \gamma\frac{k_b\,T}{\bar{m}} \quad \Rightarrow \quad V_0^2\,c'^2 = \gamma\,\frac{k_b\,\theta_0\,T'}{m_h\,\bar{\mu}} \quad \Rightarrow \quad c'^2 = \gamma\,\underbrace{\left\{\frac{k_b\,\theta_0}{m_h\,V_0^2}\right\}}_{1}\,\frac{T'}{\bar{\mu}} \tag{15}$$

$$\frac{k_b\,\theta_0}{m_h\,V_0^2} = 1 \quad \Rightarrow \quad \theta_0 = V_0^2\,\frac{m_h}{k_b} \tag{16}$$

11.4 Computing scaling variables in GANDALF

For consistency, GANDALF computes all scaling variables internally in SI units. However, this is not useful for astrophysical applications where the natural units to choose may be parsecs or megayears. Therefore, the internal scaling variables are split into two parts for convenience in chosing an appropriate. For a given unit `X`, the scaling factor is split into $X_0 = X_{\rm outscale}\, X_{\rm outSI}$, where

- $X_{\rm outscale}$: scale factor to convert X' to X in the user-requested units;
- $X_{\rm outSI}$: the requested unit in SI units.

To convert from the code units to the user-requested units, $X_{\rm user} = X_{\rm outscale}\, X'$. Alternatively, if you wish to convert directly to SI units, then you must also multiply by $X_{\rm outSI}$, i.e. $X_{\rm user} = X_{\rm outscale}\, X_{\rm outSI}\, X'$. One alternative possibility is if the user wishes to convert from code units to c.g.s. units; in which case, they must use $X_{\rm outcgs}$ in place of $X_{\rm outSI}$, i.e. $X_{\rm user} = X_{\rm outscale}\, X_{\rm outcgs}\, X'$

11.5 Converting initial conditions to code units

Converting from physical units to code units (e.g. when setting up initial conditions) is trivial once the scaling factors have been computed. Assuming the initial conditions for all variables are in the same units as specified in the parameters file, then we must simply divide by the $X_{\rm outscale}$, i.e.

$$X' = \frac{X_{\rm user}}{X_{\rm outscale}}\,. \tag{17}$$

If the variable (or parameter) is in either SI or cgs units, then they can be converted directly also using the SI or cgs scaling factors. For example, to convert a cgs quantity to code units,

$$X' = \frac{X_{\rm user}}{X_{\rm outscale}\, X_{\rm outcgs}}\,. \tag{18}$$

Inside the code, each unit is its own separate class defined in the src/Headers/SimUnits.h and src/Common/SimUnits.cpp files. All units are then stored in a single container class, called 'SimUnits'. This is then passed around the simulation as an object with the name `simunits`. The object name for each individual unit itself is usually a single character (e.g. `r` for length, `m` for mass, `t` for time, etc.. See the src/Headers/SimUnits.h files for a full list). For example, the following line could be used to convert the particle mass from physical (user) units to code units,

```
part.m /= simunits.m.outscale;
```

For converting from say cgs units to code units, we could write

```
part.m /= (simunits.m.outscale*simunits.m.outcgs);
```

11.6 Input and output units

GANDALF does support the ability to read in initial conditions in a different set of units to use specified in the parameters file. At the moment, this is only implemented in the `sf/seren_formatted` and `su/seren_unformatted`. In practice, this is implemented by noting that the input and output scaling factors should be exacty the same, i.e.

$$X_0 = X_{\rm outscale}\, X_{\rm outSI} = X_{\rm inscale}\, X_{\rm inSI} \tag{19}$$

12 To-do list

12.1 Known bugs

List of known bugs as of version 0.4.0 .

- Saitoh & Makino (2012) SPH has not been fully updated since recent code refactoring.
- For very short simulations, plots may not be updated correctly since the simulation process finishes before the plotting process has recieved its commands.
- If running a simulation in interactive mode and a different simulation is loaded into memory, then it is no longer possible to continue running that simulation.
- Rendered images are technically not done correct if smoothing lengths are smaller than the grid size (which can often be the case, although the images are fine for viewing/movie purposes).
- Rendered images do not correctly calculate automatic limits; must be inserted manually using the limit command.

12.2 Proposed features

List of possible new features for future versions.

- Finish implementation of MPI
- Allow ability to add titles to graphs in matplotlib
- Include analytical solutions for Sedov blast-wave test and freefall collapse test
- Plot star/sink-based statistics, e.g. sink mass-functions, binary statistics
- Include analysis routines for python for N-body specific statistics, e.g. Q-parameter, λ-parameter, etc..
- Strict energy-error checking for N-body simulations
- More sanity-checking, error-trapping and assert statements (to prevent crashing on erroneous input and to help debugging purposes).

A Command reference for the python functions

facade.**ListFunctions**()
List the available functions defined in facade

facade.**newsim**(*paramfile=None*, *ndim=None*, *sim=None*)
Create a new simulation object. Need to specify either the parameter file, or the number of dimensions and the simulation type. Note that it is not possible to change the number of dimensions afterwards or simulation type afterwards.

facade.**loadsim**(*run_id*, *fileformat=None*, *buffer_flag='cache'*)
Given the run_id of a simulation, reads it from the disk. Returns the newly created simulation object.

Parameters **run_id** (*str*) – Simulation run identification string.

Keyword Arguments

- **fileformat** – Format of all snapshot files of simulation.
- **buffer_flag** – Record snapshot data in simulation buffer.

facade.**run**(*no=None*)
Run a simulation. If no argument is given, run the current one; otherwise queries the buffer for the given simulation number. If the simulation has not been setup, does it before running.

Keyword Arguments **no** (*int*) – Simulation number

facade.**plot**(*x*, *y*, *type='default'*, *snap='current'*, *sim='current'*, *overplot=False*, *autoscale=False*, *xunit='default'*, *yunit='default'*, *xaxis='linear'*, *yaxis='linear'*, ***kwargs*)
Plot particle data as a scatter plot. Creates a new plotting window if one does not already exist.

Parameters

- **x** (*str*) – Quantity on the x-axis.
- **y** (*str*) – Quantity on the y-axis.

Keyword Arguments

- **type** – The type of the particles to plot (e.g. 'star' or 'sph').
- **snap** – Number of the snapshot to plot. Defaults to 'current'.
- **sim** – Number of the simulation to plot. Defaults to 'current'.
- **overplot** (*bool*) – If True, overplots on the previous existing plot rather than deleting it. Defaults to False.
- **autoscale** – If True, the limits of the plot are set automatically. Can also be set to 'x' or 'y' to specify that only one of the axis has to use autoscaling. If False (default), autoscaling is not used. On an axis that does not have autoscaling turned on, global limits are used if defined for the plotted quantity.
- **xunit** (*str*) – Specify the unit to use for the plotting for the quantity on the x-axis.
- **yunit** (*str*) – Specify the unit to use for the plotting for the quantity on the y-axis.
- ****kwargs** – Extra keyword arguments will be passed to matplotlib.

Returns Data plotted. The member x_data contains data on the x-axis and the member y_data contains data on the y-axis

facade.**addplot**(*x*, *y*, ***kwargs*)
Thin wrapper around plot that sets overplot to True. All the other arguments are the same. If autoscale is not explicitly set, it will be set to False to preserve the existing settings.

Parameters

- **x** (*str*) – Quantity on the x-axis.
- **y** (*str*) – Quantity on the y-axis.

Keyword Arguments See documentation of the plot function.

facade.**render**(*x*, *y*, *render*, *snap='current'*, *sim='current'*, *overplot=False*, *autoscale=False*, *autoscalerender=False*, *coordlimits=None*, *zslice=None*, *xunit='default'*, *yunit='default'*, *renderunit='default'*, *res=64*, *interpolation='nearest'*, *lognorm=False*, *type='sph'*, ***kwargs*)

Create a rendered plot from selected particle data.

Parameters

- **x** (*str*) – Quantity on the x-axis.
- **y** (*str*) – Quantity on the y-axis.
- **render** (*str*) – Quantity to be rendered.

Keyword Arguments

- **snap** – Number of the snapshot to plot. Defaults to 'current'.
- **sim** – Number of the simulation to plot. Defaults to 'current'.
- **overplot** (*bool*) – If True, overplots on the previous existing plot rather than deleting it. Defaults to False.
- **autoscale** – If True, the coordinate limits of the plot are set automatically. Can also be set to 'x' or 'y' to specify that only one of the axis has to use autoscaling. If False (default), autoscaling is not used. On an axis that does not have autoscaling turned on, global limits are used if defined for the plotted quantity.
- **autoscalerender** – Same as the autoscale, but for the rendered quantity.
- **coordlimits** – Specify the coordinate limits for the plot. In order of precedence, the limits are set in this way:
  - What this argument specifies. The value must be an iterable of 4 elements: (xmin, xmax, ymin, ymax).
  - If this argument is None (default), global settings for the quantity are used.
  - If global settings for the quantity are not defined, the min and max of the data are used.
- **zslice** (*float*) – z coordinate of the slice when doing a slice rendering. Default is None, which produces a column-integrated plot. If you set this variable, instead a slice rendering will be done.
- **xunit** (*str*) – Specify the unit to use for the plotting for the quantity on the x-axis.
- **yunit** (*str*) – Specify the unit to use for the plotting for the quantity on the y-axis.
- **renderunit** (*str*) – Specify the unit to use for the plotting for the rendered quantity.
- **res** – Specify the resolution. Can be an integer number, in which case the same resolution will be used on the two axes, or a tuple (e.g., (xres, yres)) of two integer numbers, if you want to specify different resolutions on the two axes.
- **interpolation** – Specify the interpolation to use. Default is nearest, which will show the pixels of the rendering grid. If one wants to smooth the image, bilinear or bicubic could be used. See pyplot documentation for the full list of possible values.

- **lognorm** (*bool*) – Specify wheter the colour scale should be logarithmic (default: linear). If you want to customise the limits, use the vmin and vmax flags which are passed to matplotlib
- **type** (*str*) – Specify the type of particles to be used for rendering (defaults to sph)
- ****kwarg** – Extra keyword arguments will be passed to matplotlib.

Returns Data plotted. The member render_data contains the actual image (2d array).

`facade.`**`addrender`**(*x*, *y*, *renderq*, ***kwargs*)

Thin wrapper around render that sets overplot to True. If autoscale is not explicitly set, it will be set to False to preserve the existing settings.

Parameters

- **`x`** (*str*) – Quantity on the x-axis.
- **`y`** (*str*) – Quantity on the y-axis.
- **`renderq`** (*str*) – Quantity to be rendered.

Keyword Arguments **See documentation of the render function.**

`facade.`**`renderslice`**(*x*, *y*, *renderq*, *zslice*, ***kwargs*)

Thin wrapper around render that does slice rendering.

Parameters

- **`x`** (*str*) – Quantity on the x-axis.
- **`y`** (*str*) – Quantity on the y-axis.
- **`renderq`** (*str*) – Quantity to be rendered.
- **`zslice`** (*float*) – z-coordinate of the slice.

Keyword Arguments **See documentation of the render function.**

`facade.`**`addrenderslice`**(*x*, *y*, *renderq*, *zslice*, ***kwargs*)

Thin wrapper around renderslice that sets overplot to True. If autoscale is not explicitly set, it will be set to False to preserve the existing settings.

Parameters

- **`x`** (*str*) – Quantity on the x-axis.
- **`y`** (*str*) – Quantity on the y-axis.
- **`renderq`** (*str*) – Quantity to be rendered.
- **`zslice`** (*float*) – z-coordinate of the slice.

Keyword Arguments **See documentation of the render function.**

`facade.`**`plotanalytical`**(*x=None*, *y=None*, *ic='default'*, *snap='current'*, *sim='current'*, *overplot=True*, *autoscale=False*, *xunit='default'*, *yunit='default'*, *time='snaptime'*)

Plots the analytical solution. Reads the problem type from the 'ic' parameter and plots the appropriate solution if implemented. If no solution exists, then nothing is plotted.

Keyword Arguments

- **x** (*str*) – Quantity on the x-axis.
- **y** (*str*) – Quantity on the y-axis.
- **snap** – Number of the snapshot to plot. Defaults to 'current'.
- **sim** – Number of the simulation to plot. Defaults to 'current'.

- **overplot** (*bool*) – If True, overplots on the previous existing plot rather than deleting it. Defaults to False.
- **autoscale** – If True, the limits of the plot are set automatically. Can also be set to 'x' or 'y' to specify that only one of the axis has to use autoscaling. If False (default), autoscaling is not used. On an axis that does not have autoscaling turned on, global limits are used if defined for the plotted quantity.
- **xunit** (*str*) – Specify the unit to use for the plotting for the quantity on the x-axis.
- **yunit** (*str*) – Specify the unit to use for the plotting for the quantity on the y-axis.
- **time** – Plots the analytical solution for the given time. If not set, then reads the time from the sim or snapshot

Returns Data plotted. The member x_data contains data on the x-axis and the member y_data contains data on the y-axis

`facade.time_plot`(*x*, *y*, *sim='current'*, *overplot=False*, *autoscale=False*, *xunit='default'*, *yunit='default'*, *xaxis='linear'*, *yaxis='linear'*, *idx=None*, *idy=None*, *id=None*, *typex='default'*, *typey='default'*, *type='default'*, ***kwargs*)

Plot two quantities as evolved in time one versus the another. Creates a new plotting window if one does not already exist.

Parameters

- **x** (*str*) – Quantity on x-axis. The quantity is looked up in the quantities defined as a function of time. If it is not found there, then we try to interpret it as a quantity defined for a particle. In this case, the user needs to pass either idx either id to specify which particle he wishes to look-up.
- **y** (*str*) – Quantity on y-axis. The interpretation is like for the previous argument.

Keyword Arguments

- **sim** – Number of the simulation to plot. Defaults to 'current'.
- **overplot** (*bool*) – If True, overplots on the previous existing plot rather than deleting it. Defaults to False.
- **autoscale** – If True, the limits of the plot are set automatically. Can also be set to 'x' or 'y' to specify that only one of the axis has to use autoscaling. If False (default), autoscaling is not used. On an axis that does not have autoscaling turned on, global limits are used if defined for the plotted quantity.
- **xunit** (*str*) – Specify the unit to use for the plotting for the quantity on the x-axis.
- **yunit** (*str*) – Specify the unit to use for the plotting for the quantity on the y-axis.
- **idx** (*int*) – id of the particle to plot on the x-axis. Ignored if the quantity given (e.g., com_x) does not depend on the id.
- **idy** (*int*) – same as previous, on the y-axis.
- **id** (*int*) – same as the two previous ones. To be used when the id is the same on both axes. If set, overwrites the passed idx and idy.
- **typex** (*str*) – type of particles on the x-axis. Ignored if the quantity given does not depend on it
- **typey** (*str*) – as the previous one, on the y-axis.
- **type** (*str*) – as the previous ones, for both axis at the same time. If set, overwrites typex and typey.

Returns Data plotted. The member x_data contains data on the x-axis and the member y_data contains data on the y-axis

facade.**savefig**(*name*)
Save the current figure with the given name. Note that matplotlib figures out automatically the type of the file from the extension.

Parameters **name** (*str*) – filename (including extension)

facade.**next**()
Advances the current snapshot of the current simulation. Return the new snapshot, or None if the call failed.

facade.**previous**()
Decrements the current snapshot of the current simulation. Return the new snapshot, or None if the call failed.

facade.**snap**(*no*)
Jump to the given snapshot number of the current simulation. Note that you can use standard Numpy index notation (e.g., -1 is the last snapshot). Return the new snapshot, or None if the call failed.

Parameters **snapno** (*int*) – Snapshot number

Returns The snapshot object

facade.**block**(*message='Press enter to quit...'*)
Stops the execution flow until the user presses 'enter'. Useful in scripts, allowing to see a plot (which otherwise gets closed as soon as the execution flow reaches the end of the script)

Keyword Arguments **message** (*str*) – text to print before pausing

facade.**sims**()
Print a list of the simulations to screen

facade.**snaps**(*simno*)
For the given simulation number, print a list of all the snapshots

Parameters **simno** (*int*) – Simulation number from which to print the snapshot list.

facade.**set_current_sim**(*simno*)
Set the current simulation to the given number.

Keyword Arguments **simno** (*int*) – Simulation number

Returns The newly set current simulation

facade.**limit**(*quantity*, *min=None*, *max=None*, *auto=False*, *window='current'*, *subfigure='current'*)
Set plot limits. Quantity is the quantity to limit.

Parameters **quantity** (*str*) – Set limits of this variable.

Keyword Arguments

- **min** (*float*) – Minimum value of variable range.
- **max** (*float*) – Maximum value of variable range.
- **auto** (*bool*) – If auto is set to True, then the limits for that quantity are set automatically. Otherwise, use the one given by max and min.
- **window** (*str*) – By default only the current subplot of the current window is affected. If this parameter is set to 'all', all the current windows are affected. If this parameter is set to 'global', then also future plots are affected.
- **subfigure** (*str*) – Similarly to window, by default only the current subplot is affected by this command. If this parameter is set to 'all' then all the subfigures in the current window are affected.

facade.**make_movie**(*filename*, *snapshots='all'*, *window_no=0*, *fps=24*)

Generates movie for plots generated in given window

Parameters

- **filename** (*str*) – filename (with extension, e.g. mp4) of the movie that will be created.
- **snapshots** (*str*) – currently not used
- **window_no** (*int*) – currently not used
- **fps** (*int*) – frames per second

facade.**KnownQuantities**()

Return the list of the quantities that are defined

facade.**get_data**(*quantity*, *snap='current'*, *type='default'*, *sim='current'*, *unit='default'*)

Returns the array with the data for the given quantity. The data is returned scaled to the specified unit

Parameters **quantity** (*str*) – The quantity required.

Keyword Arguments

- **type** (*str*) – The type of the particles (e.g. 'star')
- **snap** – Number of the snapshot. Defaults to 'current'
- **sim** – Number of the simulation. Defaults to 'current'
- **unit** (*str*) – Specifies the unit to use to return the data

Returns A numpy array containing the requested data.

facade.**get_render_data**(*x*, *y*, *quantity*, *sim='current'*, *snap='current'*, *renderunit='default'*, *res=64*, *zslice=None*, *coordlimits=None*)

Return the rendered data for the given quantity. Useful when one needs to grid SPH data. The result is scaled to the specified unit. The options are a subset of the options available to the 'render' function.

Parameters

- **x** (*str*) – Quantity on the x-axis.
- **y** (*str*) – Quantity on the y-axis.
- **quantity** (*str*) – Quantity to render.

Keyword Arguments

- **snap** – Number of the snapshot to plot. Defaults to 'current'.
- **sim** – Number of the simulation to plot. Defaults to 'current'
- **renderunit** (*quantity*) – Unit to use for the rendered quantity
- **res** – Resolution
- **zslice** (*float*) – z-coordinate of the slice when doing a slice rendering. Default is None, which produces a column-integrated plot. If you set this variable, a slice rendering will be done instead.
- **coordlimits** – Limits of the coordinates on x and y. See documentation of render.

Returns A numpy 2d array containig the rendered data, scaled to the requested unit.

facade.**CreateTimeData**(*name*, *function*, **args*, ***kwargs*)

Given a function that takes a snapshot as input, construct a new quantity which can be used to do time plots. Technically this implemented by constructing a new object of type FunctionTimeDataFetcher. See userguide for how to use this function.

Parameters

- **name** (*str*) – the name of the new quantity
- **function** – a python function that computes the desidered quantity. See the userguide for examples.
- ****kwargs** – extra keyword arguments will be passed to your function (see example of lagrangian_radii)

Returns The FunctionTimeDataFetcher object newly constructed.

facade.**CreateUserQuantity**(*name*, *formula*, *unitlabel=''*, *unitname=''*, *scaling_factor=1*, *label=''*)

Create a new quantity that can be used for example for plots. This can be done both by giving a mathematical formula, or providing a user-defined function. The quantity is given a name, which can now be used in plots and in other formulae. When you construct a quantity, you can rely on one of the units we provide, in which case you can just pass as the scaling_factor parameter the name of the unit you want inside the SimUnits class. For example, if your unit has dimensions of acceleration, you can pass 'a' as the scaling_factor parameter. Doing this allows the unit system to work seamlessly when plotting (i.e., you can specify the units you want the plot in). Alternatively, you can build your own unit passing a numerical value for the scaling_factor, a unitname and a latex label. In this case, however, no rescaling is possible, as the unit system does not know how to rescale your unit. Technically, this function works by constructing either a FormulaDataFetcher object or a FunctionFetcher object.

Parameters

- **name** (*str*) – the name of the new quantity created
- **formula** – either a string with the mathematical formula, or a user defined function that computes the desired quantity. See the userguide for further reference.
- **unitlabel** (*str*) – label of the unit (to use in plots in the axis names)
- **unitname** (*str*) – name of the unit (to be passed to functions that take a unit keyword)
- **scaling_factor** – either a string with a quantity with the same dimension, or a float that will be used to multiply the result of the formula/function
- **label** (*str*) – latex label of the quantity to be used on the axis when plotting

Returns the newly constructed object which represents the desired quantity

facade.**window**(*no=None*)

Changes the current window to the number specified. If the window doesn't exist, recreate it.

Parameters **winno** (*int*) – Window number

facade.**subfigure**(*nx*, *ny*, *current*)

Creates a subplot in the current window.

Parameters

- **nx** (*int*) – x-grid size
- **ny** (*int*) – y-grid size
- **current** (*int*) – id of active sub-figure. If sub-figure already exists, then this sets the new active sub-figure.

facade.**switch_nongui**()

Switches matplotlib backend, disabling interactive plotting. Useful in scripts where no interaction is required

facade.**rescale**(*quantity*, *unitname*, *window='current'*)

Rescales the specified quantity in the specified window to the specified unit

Parameters

- **quantity** (*str*) – Quantity to be rescaled.

- **unitname** (*str*) – Required unit for quantity.

Keyword Arguments **window** – Window containing plot. Can be either the string “current” or an integer specifying the window.

Python Module Index

f