CONCEPT

The COsmological N-body CodE in PyThon

Jeppe Mosgaard Dakin
jeppe.mosgaard.dakin(at)post.au.dk

October 12, 2015

This document serves as a user's guide to the CONCEPT code. It is split up into two parts, the first of which gives a basic overview of the functionality of the program, while the second is much more in-depth.
The GitHub page for the CONCEPT code can be found at
https://github.com/jmd-dk/concept/

Contents

Part I

Overview

1 Introduction

CONCEPT (COsmological N-body CodE in PyThon) is a free and open-source code for cosmological N-body simulations on massively parallel computers with distributed memory. As of the time of writing, collisionless dark matter is the only implemented particle species. Gravity can be computed using the PP, PM or the P^3M algorithm. The goal of CONCEPT is to make it pleasant to work with cosmological N-body simulations — for the cosmologist as well as for the source code developer.

CONCEPT is easy to use and has good performance. The source code is properly structured, documented (and well-commented) and consist almost exclusively of PEP 8 compliant* Python code.

CONCEPT relies on the Cython compiler, which translates (or *cythonizes*) Python source code to equivalent C source code, which are then compiled as any other C program. The computationally expensive parts of the code are written such that they translate very directly into C code, while the inexpensive parts are written using the whole spectrum of modern Python constructs, which are only poorly translated into C code†. In effect this allow for high performance *and* rapid development.

In addition to running CONCEPT in compiled mode, it is also possible to run it as any other Python script, though with a hefty performance penalty.

The CONCEPT code is written in version 3 of the Python language. Among other things, this allows for Unicode literals in strings and variable names, which CONCEPT takes advantage of. All source files are therefore encoded in UTF-8. This unconventional and slightly dangerous choice is made in the name of æsthetics.

Although CONCEPT is intended for computer clusters, it really runs on anything down to single-core laptops. Communication between CPUs is done explicitly by means of the message passing interface (MPI). The code has been tested and found to work on a large number of Linux systems.

*PEP 8 is the official style guide for writing clean Python code.

†When no equivalent C construct exists for a given Python construct, Cython simply generates the CPython C API calls corresponding to the given Python construct.

1.1 Jargon

The particles of the simulation lives within a periodic box which constitutes the simulated universe. A file containing the total state of the system (the positions and momenta of all N particles) at any given time is referred to as a *snapshot*. The initial snapshot constituting the initial conditions is then called the *initial condition file*, or *IC file* for short. Besides the initial conditions, a particular simulation is defined by the values of a set of parameters, controlling the cosmology, physics and numerical schemes and accuracies of the simulation. A file containing the collective set of parameter values is called a *parameter file*. The user-defined input to each simulation is then an IC file and a parameter file. A particular instance of a simulation is referred to as a *run*.

2 Basic Usage

Here we state the very basic functionalities of CONCEPT in a tutorial-like manner*. For a much more in-depth description, see part II of this document.

2.1 Installation

To install CONCEPT along with all the dependency programs it needs, simply run the command

```
bash <(wget -O- --no-ch bit.ly/concept_nbody)
```

from a terminal, which will download and launch the CONCEPT installer. You will be prompted for an installation directory in which everything will be placed. To uninstall CONCEPT, simply delete this directory. The installation will consume slightly less than 1 GB of disk space and will complete within 1–2 hours on modern hardware.

2.2 Files and Directories

A stripped down version of CONCEPT's directory tree is shown in figure 2.1. The `concept` directory contain four subdirectories and the `concept` executable, which is just a Bash script. The `ICs` subdirectory is intended for storage of IC files. As part of the CONCEPT distribution, an example IC file is provided. This IC file is written in CONCEPT's so-called "standard" HDF5 format. Similarly to `ICs`, the `params` subdirectory is intended for the storage of parameter files. An example parameter file, which is just a text file of definitions, is provided. Parameter and IC files are described in detail in chapters 5 and 6, respectively.

The `output` directory is the default directory for CONCEPT to dump its output, while `logs` is where it save log files. Everything printed to the screen in each run will be logged automatically.

*If this is your first experience with CONCEPT, it is advised that you try it out on a local computer rather than on a remote host, as the behaviour of CONCEPT differ somewhat for these two cases.

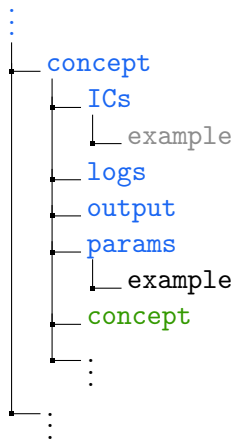

Figure 2.1 – Basic directory tree for CONCEPT. The colors indicate directories (blue), Bash scripts (green), text files (black) and HDF5 files (gray).

2.3 Running the Code

The simplest possible invocation of CONCEPT — assuming that you are in the `concept` directory — is

```
./concept
```

You will be told that you did not specify neither the number of processes nor the parameter file to use. CONCEPT then resorts to running a single process and using its internal default values for all parameters. If this is the first run or the code is not build for some other reason, CONCEPT will now do so. When the build process is completed, the actual run will begin and immediately finish, without really doing anything.

To control the number of processes to use and specify a parameter file, invoke CONCEPT like e.g.

```
./concept -n 2 -p params/example
```

which will use two processes to run the simulation defined by the example parameter file. As ordered by the parameter file, a snapshot will be produced in the `output` directory.

The `concept` script has many more options, which will be described in detail in section 4.2. For now you may take a quick look at the available options using

```
./concept -h
```

2.3.1 Parameters and Output

Let us take a short look at some of the parameters in the example parameter file. Opening the file we see that the parameters are grouped into categories. In

the first category, input/output, the IC file together with any wanted output is specified. Below some of the parameters in this category are given new values:

```
# Input/output
IC_file = 'ICs/example'
snapshot_type = 'standard'
output_dirs = {
    'snapshot' : 'output',
    'powerspec': 'output/powerspec',
    'render'   : '',
               }
output_bases = {
    'snapshot' : 'spam',
    'powerspec': 'ham',
    'render'   : 'eggs',
                }
output_times = {
    'snapshot'       : 1,
    'powerspec'      : (0.5, 1),
    'render'         : linspace(0.05, 1, 5),
    'terminal render': [sqrt(i) for i in (0.3, 0.4, 0.5)],
                }
```

As listed in the `output_times` variable, four different kinds of output are available:

Snapshots
: This is simply a dump of the raw state of the particle system, in the format specified by `snapshot_type`.

Power spectra
: Instead of simply outputting the raw data, you may be more interested in some reduced form of it. The power spectrum is one such form. The primary power spectrum output is a text file containing the power spectrum data. Optionally, CONCEPT is able to also plot the power spectrum and produce an image file (look for the `powerspec_plot` parameter in the graphics category). Throughout the CONCEPT code, a power spectrum is referred to as a *powerspec*.

Renders
: CONCEPT is able to produce 3D graphical visualizations of the particles. Each render output is an image of one such visualization. The appearance of the renders can be adjusted using parameters in the graphics category.

Terminal renders
: Like renders, terminal renders are graphical visualizations of the particles. Unlike renders however, terminal renders get printed directly in the terminal and are not saved to disk (disregarding the log file). The

appearance of the terminal renders can be adjusted using parameters in the graphics category.

Since all four kinds of output have non-empty output times in the parameter file snippet above, they are all going to be produced. An output will be produced during the simulation when the scale factor matches a value given in `output_times`. As demonstrated above, CONCEPT accepts values for `output_times` in a variety of different formats. It holds generally for all parameters that any meaningful and valid Python statement may be used as their definition.

The values in `output_dirs` are the directories in which to dump the different kinds of output. These — as well as all other relative paths in the CONCEPT code — should be stated relative to the `concept` directory. In the parameter file snippet above, we see that snapshots will be placed in the standard `output` directory, while power spectra will be placed in the non-standard `output/powerspec` directory. Any non-existing output directories will be created as needed. Lastly, we see that no output directory is assigned to renders. As renders are going to be produced, CONCEPT assigns `output` as the render output directory. This is an example of CONCEPT's use of default parameter values. You can leave out any parameter from your parameter files and they will be handled for you in a similar manner. It is however not generally advised to write parameter files which depend on the choice of such default assignments.

The values in `output_bases` serves as filename prefixes given to the different output. This together with the output time labels the output files. The parameter file snippet above will e.g. produce the output files `output/spam_a=1.00`, `output/powerspec/spam_a=1.00` and `output/eggs_a=1.00.png`.

Part II

In-depth Guide

3 Installation

The CONCEPT code should be able to compile and run on virtually any Linux system. The software stack needed to build and run the code is quite large, taking up nearly 1 GB of disk space. To ease the process of installation of this stack, CONCEPT comes with an `installer` script, which automates this process.

Some basic system dependencies* are required to run even the `installer`. These programs are pre-installed on most Linux systems. If `installer` is run while some system dependencies are missing, these will be reported to you. Additionally, if the `installer` finds an installed package-manager, it will prompt for installation of the missing system dependencies (requiring root privileges).

3.1 Dependencies

The CONCEPT code has the following free and open-source software dependencies:

GSL The GNU Scientific Library.

MPI Any implementation of the Message Passing Interface (version 3.0 or newer).

HDF5 The Hierarchical Data Format, linked to the MPI library and configured to be parallel. It has the zlib library as a dependency.

FFTW The Fastest Fourier Transform in the West (version 3.3 or newer), linked to the MPI library and configured to be parallel.

Python The standard C-implementation of the Python programming language (version 3.3 or newer). For pip† to be installed as part of Python, Python

*By "system dependencies" we mean software components which are usually installed system-wide by the root user. The complete list of system dependencies are AWK, Bash (version 4.0 or newer), sed, GNU Coreutils, GNU Make, GCC, GFortran, G++, gzip, Perl, tar and wget. Only the first six of these are strictly necessary to run the CONCEPT code, though all of them are needed to run the `installer` script.

†The Python package manager pip is handy when it comes to installing Python packages.

needs to be build against the OpenSSL and zlib libraries. The Python package Blessings depends on the _curses module, which is installed as part of Python provided that Python is build against the ncurses library. The following* Python site-packages are required:

Blessings This package depends on the _curses Python module.

Cython Version 0.22.0 or newer, though not 0.23.

Cython_GSL This needs to be linked to the GSL library.

H5Py Version 2.4 or newer, linked to the HDF5 library and configured to be parallel.

Matplotlib This package depends on the FreeType library.

MPI4Py Version 1.3.1 or newer, linked to the MPI library.

NumPy

Pexpect

ImageMagick This is only needed for producing renders when running with more than 1 CPU and can be omitted. ImageMagick needs to be build with png and zlib features, which require the pnglib and zlib libraries.

GADGET The GAlaxies with Dark matter and Gas intEracT N-body code (version 2.0.7). This is only needed for running some tests and can be omitted. GADGET is also dependent on GSL, MPI and FFTW. Note that GADGET is incompatible with FFTW 3.x, so a separate FFTW 2.x must also be installed.

3.2 The `installer`

The CONCEPT code and all of the dependencies listed in the previous section (including the dependencies of the dependencies ...) can be installed and linked together automatically by running the `installer` Bash script. You can find the script in the GitHub repository†, or you can simply run‡§

```
bash <(wget -O- --no-ch bit.ly/concept_nbody) [install_dir]
```

to automatically fetch and execute the script. Here, `install_dir` is the path to the directory in which CONCEPT should be installed. If you leave out this path, you will be prompted for it. Naturally, it is also possible to save the `installer` to disk and invoke it like e.g.

*Some of these depend on other Python packages which are not listed. If installing via pip, these additional Python packages will automatically be installed.

†https://raw.githubusercontent.com/jmd-dk/concept/master/installer

‡The `bit.ly/concept_nbody` part refers to the URL http://bit.ly/concept_nbody, which redirects to the `installer` script on GitHub. If the short version does not work, use the full URL.

§The square bracket indicates an optional argument.

```
./installer [install_dir]
```

The installer script will use MPICH for the needed MPI implementation*. Besides the listed Python packages and their own Python package dependencies, the installer will install the pip and Yolk3k Python packages, which is used to install the primary packages.

The installer script will install CONCEPT and the entire dependency stack within the chosen installation directory. Uninstalling everything therefore simply amounts to removing this directory.

On modern hardware, it takes the installer around 1–2 hours to complete the installation. This is mainly because the installer script insists on installing everything from source, rather than using pre-compiled binaries. One benefit from this choice is that the installed software become tailor-made for the hardware in question, optimizing the performance. The primary reason for this choice is however portability (e.g. the installer is able to install CONCEPT and the dependency stack on both 32-bit and 64-bit architectures).

A significant fraction of the installation time is spend running the test suites that the many programs provide. To speed up the installation (and leave potential errors unnoticed!), you may skip these tests by invoke the installer with the --fast option:

```
# from local copy of installer
./installer --fast [install_dir]
```

```
# from the installer on GitHub
bash <(wget -O- --no-ch bit.ly/concept_nbody) --fast [install_dir]
```

To decrease the installation time further, the --fast option also sets any idle time to zero, decreasing the chances for successful downloading for questionable connections.

3.2.1 Using Pre-installed Libraries

By default, any pre-installed versions of the above software will be ignored. Should you wish the installer to use such pre-installed components, you have to declare their directory paths through *name*_dir environment variables, where *name* is the name† of the pre-installed program. E.g. to use a pre-installed GSL library, invoke the installer like so:

*Note on possible problems related to using MPICH on clusters with several nodes: MPICH uses ssh to launch MPI-threads on remote nodes, which require that password-less login to the cluster nodes is enabled (this is not to be confused with password-less login to the cluster itself). Should you encounter problems when running multi-node CONCEPT jobs, this may be the problem. In addition, newer version of ssh blocks password-less login if your home-directory has group write access.

†The valid *name*s are exactly those of the dependencies listed in section 3.1, written in all lowercase. For the FFTW 2.x library used by GADGET, use fftw_for_gadget_dir.

```
# from local copy of installer
gsl_dir=/path/to/gsl ./installer [···]
```

```
# from the installer on GitHub
gsl_dir=/path/to/gsl bash <(wget -O- --no-ch bit.ly/concept_nbody) [···]
```

Here and in what follows, `[···]` stands for a space-seperated list of optional arguments (e.g. `install_dir` and `--fast`). Multiple program paths are specified as a space-separated list.

Should you choose to use a pre-installed Python distribution, it needs to satisfy the following requirements:

- The pip Python package needs to be installed if any of the required Python packages are not installed.
- The _curses Python module needs to be installed.

Pre-installed Python packages are automatically detected when the `python_dir` environment variable is set. That is, you do not have to set e.g. `blessings_dir`.

3.2.2 Requesting Specific Versions

Should you wish to install a specific version of some dependency, simply set the environment variable *name*`_version` when invoking the installer. This also works for Python packages. E.g. to install version 1.15 of GSL:

```
# from local copy of installer
gsl_version=1.15 ./installer [···]
```

```
# from the installer on GitHub
gsl_version=1.15 bash <(wget -O- --no-ch bit.ly/concept_nbody) [···]
```

Multiple program versions are specified as a space-separated list.

3.3 Manual Installation

To do a manual installation of any of the dependency programs, we refer to their individual online documentation.

To install just the CONCEPT code without any of the dependency programs, simply download the CONCEPT code from its GitHub repository* and extract it into some directory. To make CONCEPT aware of the installed dependency programs, simply edit the `.paths` file, which you will find in the topmost directory. The `.paths` file contains the *absolute* paths to every† file and directory which CONCEPT needs to know about. No path information is ever

*https://github.com/jmd-dk/concept/archive/master.tar.gz

†With the exception of system dependencies (e.g. AWK) which are expected to be on the `PATH`.

drawn from anywhere else. The description of each variable within the `.paths` file itself should constitute enough documentation of which variable should contain which path.

All the paths specified in the `.paths` file are read in at runtime by CONCEPT. Should you ever wish to make CONCEPT aware of additional paths, simply add them to the `.paths` file.

4 Running the Code

In this chapter the primary functionalities and invocation methods of CONCEPT are described. In addition, CONCEPT implements secondary functionalities — called *utilities* — which are not used to run simulations directly, but serve as convenient tools within the program environment. For these secondary uses of CONCEPT, see chapter 7.

A directory tree of CONCEPT is shown in figure 4.1. Disregarding IC and parameter files, the `concept` script is the only file that the user should ever have to interact with directly in order to run the code. In the simplest case (see the next section for the differences between local and remote execution), executing the `concept` script will build and run the code.

As compilation is performed automatically when the code is executed, before describing how to run the code in detail, a brief description of this build process is in order. When the `concept` script is executed, it invokes the `Makefile` which builds the program. The compiled program consists of a set of shared object (`.so`) files; precisely one for each of the modules (the Python `.py` files grouped together in figure 4.1). To run the program, the `main` module, `main.so`, is invoked via the Python interpreter, which itself is started by MPI. Running in so-called "pure Python mode", meaning without first having compiled the code, amounts to skipping the entire build process but otherwise invoking the `main` module (now `main.py`) in a similar fashion. For the exact commands needed to run CONCEPT manually, see section 4.3.

Each CONCEPT run will be assigned a unique ID. The combined output to `stdout` and `stderr` will always be logged to the file `logs/`*`ID`*, where *`ID`* is the ID of the current run in question. The output from `stderr` alone will similarly be saved to `logs/`*`ID`*`_err`.

4.1 Local vs. Remote Execution

Running the `concept` script on a local machine will build (if necessary) and run the code. When logged on to a remote host though, the default behaviour is to build the program and do nothing more. This behaviour is chosen in order to bring the user in control of how the program should be executed. Often, computer clusters utilizes a job scheduling mechanism, such as PBS (Portable

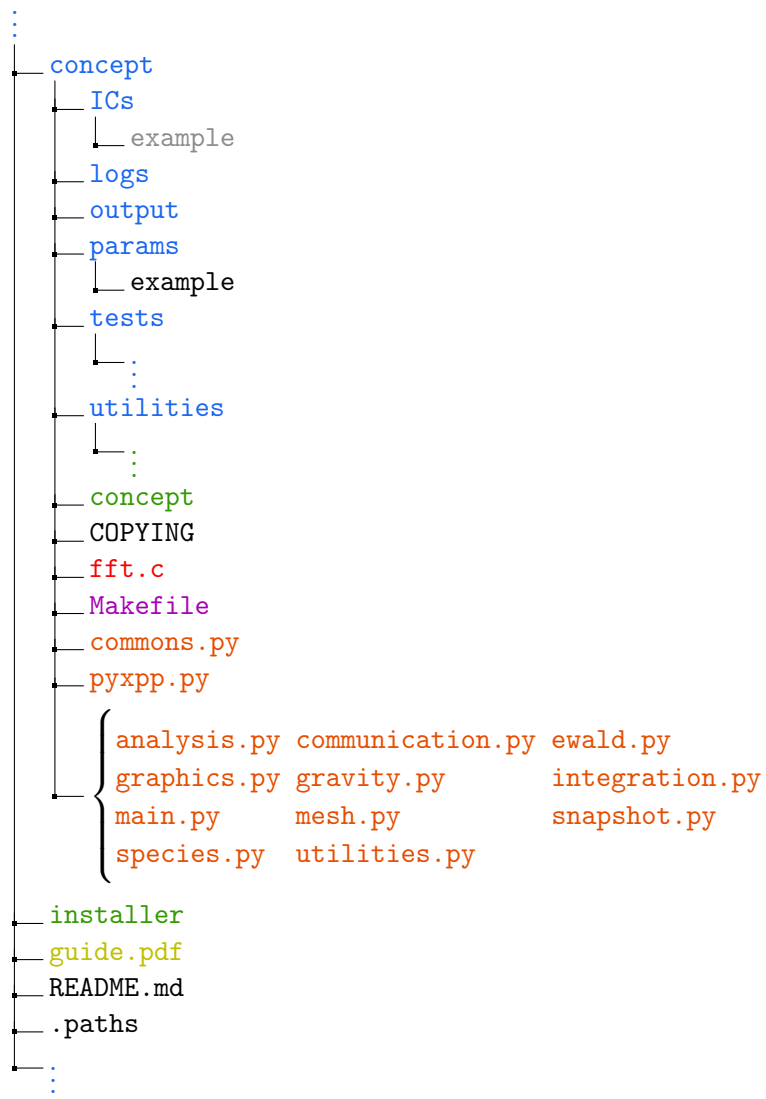

Figure 4.1 – Directory tree for CONCEPT. As in figure 2.1, the colors indicate directories (blue), Bash scripts (green), text files (black) and HDF5 files (grey). In addition we now have C source code files (red), makefiles (purple), PDF files (yellow) and Python files (orange). Two of the Python files are listed separately, while the others are grouped together. The files in this group are referred to as Python *modules*.

Batch System). The CONCEPT script *is* able to autogenerate and submit PBS job scripts*. If this feature is desired, you need to edit the first couple of lines in the `concept` script as follows:

*Even though several versions of PBS exist (such us OpenPBS, TORQUE and PBS Professional), each with slightly different functionality, the core components used by `concept` are the same in all of them.

Populate the queues and ppns variables with queue names and corresponding numbers of CPUs per node. As an example, imagine that we wish CONCEPT to be aware of the queues 'q8' and 'q4', which have 8 and 4 CPUs per node, respectively. In that case, we would set the variables as
queues=(q8 q4)
ppns=(8 4)

Now when executing the concept script on the remote host, a queue from the queues variable will be chosen for the job, based on its number of CPUs per node. To ensure maximum exploitation of the available resources, a queue will only be deemed fit for use if the number of requested processes is divisible by the number of CPUs per node in the queue (so that the job utilizes every CPU on the nodes it occupy). If no queue satisfies this requirement, the job will not be submitted. If multiple queues satisfy the requirement, the one listed first in the queues variable will be used. A job script will then be generated, saved as concept/jobscript and submitted.

Should you wish to run CONCEPT directly, without the use of PBS, even though you are not working on a local machine, invoke concept with the --local option:

```
./concept --local
```

The next section lists with the full set of options available when executing concept.

4.2 The concept Command-Line Interface

As already demonstrated, several optional options may be given when invoking the concept script. As is customary for UNIX shell scripts, a list of available options together with brief descriptions will be shown when invoking the concept script with the -h or --help option. Here follows the same list but with much extended descriptions. Remember that supplied paths may be either absolute or relative to the concept directory.

-h, --help
: Show the help message and exit.

-m MAIN, --main MAIN
: Sets the main entry point of the code to the path MAIN. The default value is concept/main.py. This option is used internally by CONCEPT when running tests, but does not currently have other use cases.

-n NPROCS, --nprocs NPROCS
: Sets the number of MPI processes to NPROCS. The default value is 1. Note that this can take on values larger than the number of available CPUs, at the cost of performance.

`-p PARAMS, --params PARAMS`
: Sets the parameter file to be used to `PARAMS`. If not set, default parameters as specified in `commons.py` are used.

`-q QUEUE, --queue QUEUE`
: Sets the queue to be used when submitting as a remote PBS job to `QUEUE`. The specified queue must be predefined in the `queues` and `ppns` variables in the top of the `concept` script. If this option is not set, the first queue in the `queues` variable which have a matching number of CPUs per node will be chosen.

`-t TEST, --test TEST`
: Instead of running a simulation, run the test `TEST`. The possible values for `TEST` are the names of the subdirectories within the `concept/tests` directory, possibly with the relative or absolute path included. In addition it is possible to run all tests by setting `TEST` to `all`. See chapter 8 for details.

`-u UTIL [···], --util UTIL [···]`
: Instead of running a simulation, run the utility `UTIL`. The possible values for `UTIL` are the names of the subdirectories within the `concept/utilities` directory, possibly with the relative or absolute path included. Most utilities require an additional argument to be passed. See chapter 7 for details.

`-w WALLTIME, --walltime WALLTIME`
: Sets the PBS walltime to `WALLTIME`, in whole hours. The default is 72 hours.

`--local`
: Build and run the code as if working on a local machine, regardless of whether this is actually the case. That is, even if working remotely, the code will be executed directly, rather than submitted as a PBS job.

`--pure-python`
: Skips the build process. The code will then resort to use the raw Python files directly. This is generally not preferred due to the severe performance penalty this entails, but it can be useful for debugging and running the code with a main entry point not suited for cythonization. If the code is already fully build, the shared object (`.so`) files will be renamed to `.so_`, in order for Python not to pick up these compiled modules. The next time `concept` is invoked without the `--pure-python` option, the compiled module files will be renamed back to `.so`.

4.3 Manual Build and Run

Should you ever need to build and run CONCEPT manually, i.e. without invoking the `concept` script, this section lists the needed commands. When running the code manually, logging is your responsibility.

To build the code, simply type

```
make
```

within the `concept` directory. When the code is fully build, you can execute it as

```
mpiexec -n NPROCS python -B -m main.so "params='PARAMS'"
```

where `mpiexec` and `python` are, respectively, the MPI executable and the Python interpreter, the paths of which should be written in the `.paths` file. As in section 4.2, `NPROCS` and `PARAMS` are the number of processes and the path to the parameter file, respectively.

Should you wish to run in pure Python mode, you first have to remove the shared object files. To remove all files resulting from compilation, type

```
make clean
```

Now to run CONCEPT in pure Python mode, execute is as

```
mpiexec -n NPROCS python -B main.py "params='PARAMS'"
```

4.4 Cleaning

Sometimes it is useful to be able to quickly remove a particular set of files. Therefore the `Makefile` defines the following clean targets:

clean
: Removes all files created by the build process.

clean_auxiliary
: Removes files generated by running CONCEPT itself, like the `jobscript` and FFTW wisdom.

clean_logs
: Removes all files in the `logs` directory.

clean_output
: Removes all files and directories in the `output` directory.

clean_tests
: Removes all files in the subdirectories of the `test` directory, which were created by running tests.

clean_utilities
Remove all files generated as a bi-product of using utilities.

distclean
Calls all of the targets above. If you yourself have not created any additional files, the CONCEPT environment is now in a distribution ready state.

5 Parameters

The parameters of a given run must be supplied in a parameter file, which is just a text file containing Python assignments. An example of such a parameter file is `concept/params/example`. In it, a short description of each parameter is written next to it. A complete description of each parameter is given below.

When assigning values to a parameter with some physical unit, the user is strongly urged to write this unit explicitly in the value. As an example consider the specification of `boxsize`, the linear size of the simulation box:

```
boxsize = 100*Mpc  # Great!
boxsize = 100      # Will probably work, but bad!
```

Here, the first definition is much preferable to the latter, even though CONCEPT by default uses Mpc as the base unit for lengths, and so numerically the definitions are equivalent. For one, having implicit units floating around is not good practice. More importantly, it will lead to trouble if someone where to redefine the internal unit system used by CONCEPT, which can easily be done by changing a few lines in the Units class in `commons.py`. Should you wish to extend the set of units known by CONCEPT, this is also the place to go.

In addition to physical units, any valid Python construct are allowed in parameter definitions. All normal mathematical as well as NumPy functions and constants are also available. As always, paths may be absolute or relative to the `concept` directory. In addition, paths from the `.paths` file are accessible as e.g. `paths['concept_dir']` for the `concept` directory. The directory of the parameter file itself is accessible as `paths['params_dir']`.

Here follows a full list of parameters, grouped into five sections.

Input/output

`IC_file`
: Path to the snapshot file containing the initial conditions for the simulation. The snapshot may be in any format known to CONCEPT. The code will automatically figure out the snapshot format at load time.

`snapshot_type`
: The format of the output snapshots. The implemented snapshot formats are described in detail in chapter 6. At the time of writing,

CONCEPT implements two snapshot formats, called `'standard'` and `'GADGET 2'`, referring to the code's own HDF5 snapshot type and the secondary snapshot type defined by GADGET-2, respectively.

`output_dirs`
: This is a dictionary with keys (left of colon) corresponding to output types and values (right of colon) giving the paths to the respective output directories. That is, its general form is like the following:

```
output_dirs = {
    'snapshot' : '/path/to/snapshots',
    'powerspec': '/path/to/powerspecs',
    'render'   : '/path/to/renders',
              }
```

Output directories of snapshots, power spectra and renders are thus defined individually. If a specified directory does not exist, it will be created.

`output_bases`
: This is a dictionary of the same form as `output_dirs`. Its values refer to the basenames (prefixes) of the output files.

`output_times`
: This is a dictionary of the same form as `output_dirs`, though with the additional `'terminal render'` key. The values correspond to those values of the scale factor for which an output should be produced. A value can be a single number or any Python sequence of numbers. Leave a value empty (set it to e.g. `''`) or remote its key completely in order not to produce output of this kind.

Numerical parameters

`boxsize`
: The linear, comoving size of the cubic simulation box. The simulated, comoving volume is then $\texttt{boxsize}^3$.

`ewald_gridsize`
: Linear size of the grid of Ewald corrections. The total number of tabulated Ewald corrections is then $\texttt{ewald\_gridsize}^3$.

`PM_gridsize`
: Linear size of the grid used by the PM method as well as the power spectrum computation. The simulation volume will then be discretized into $\texttt{PM\_gridsize}^3$ grid points.

`PM_scale`
: The scale of the gravitational force splitting into a long-range component and a short-range component, used in the P^3M method. The scale should be given in units of the linear size of a PM cell. A value slightly above 1 should be optimal.

`P3M_cutoff`
: Distance beyond which short-range forces are ignored in the P^3M method. This distance should be given in units of `P3M_scale`. A value of at least 4.8 guarantees that the force components left unaccounted for constitutes less than 1 % of the total force.

`softeningfactors`
: Mapping from particle species to values of their respective gravitational softening. The values given correspond to the radius of a softening Plummer sphere, in units of the mean interparticle distance $\texttt{boxsize}/N^{1/3}$. For dark matter, a value of a couple of percent is optimal.

`Δt_factor`
: The global time step size, given in units of the instantaneous age of the universe.

`R_tophat`
: Radius of the top-hat used to compute σ, the rms density variation, from the power spectrum.

Cosmological parameters

`H0`
: The Hubble parameter at the present time ($a = 1$). If this does not match the corresponding value in the `IC_file`, a warning will be thrown.

`Ωm`
: The matter density parameter at the present time ($a = 1$). If this does not match the corresponding value in the `IC_file`, a warning will be thrown.

`ΩΛ`
: The dark energy density parameter at the present time ($a = 1$). If this does not match the corresponding value in the `IC_file`, a warning will be thrown.

`a_begin`
: The value of the scale factor at the beginning of the simulation. If this does not match the value of the scale factor in the `IC_file`, a warning will be thrown.

Graphics

`powerspec_plot`
: Boolean value determining whether or not when outputting a power spectrum (a text file of data), a plot of the data should also be made.

`color`
: The color of the particles in the renders. This can be a number in the range 0–1 (grayscale), any sequence of 3 numbers in the range 0–1 (RGB values) or a string containing any of the color names defined by Matplotlib. See http://matplotlib.org/mpl_examples/color/named_colors.hires.png for the entire repertoire.

`bgcolor`
: The background color on the renders. The format is similar to that of `color`.

`resolution`
: The height and width of the saved renders, in pixels. Note that the images will always be square.

`liverender`
: Path to the file where the latest frame should be saved, resulting in a "live" visualization of the simulation. Set an empty path to disable this feature.

`remote_liverender`
: If live render is used, this parameter allows for uploading of the live render to a remote host. Secure copy (scp) is used. The parameter takes the usual scp form; `'user@host:/path/to/liveframe'`. When using this parameter, CONCEPT will prompt you once for your password to the host. Use an empty string to disable this feature.

`terminal_colormap`
: Sets the colormap used in terminal renders. This can be any colormap defined by Matplotlib. See http://matplotlib.org/examples/color/colormaps_reference.html for a complete list.

`terminal_resolution`
: The width of the terminal renders, in characters. The height is chosen to be half as big as the width, resulting in roughly rectangular (pixelwise) terminal renders on modern terminals where each character is about twice as high as they are wide.

Simulation options

`kick_algorithms`
: Mapping from particle species to the algorithm which should be used when kicking particles of that species. Implemented kick algorithms are `'PP'`, `'PM'` and `'P3M'`.

`use_Ewald`
: This should normally be set to `True`, in which case periodic corrections to the gravitational force will be computed via Ewald summation, when using the PP method. Otherwise, no Ewald corrections will be computed.

`fftw_rigor`
: The "rigor level" used when acquiring FFTW wisdom. It can be any of `'estimate'`, `'measure'`, `'patient'` or `'exhaustive'`. For the official FFTW documentation about each rigor level, see http://www.fftw.org/doc/Planner-Flags.html. If FFTW wisdom of the right kind but for a higher level rigor than requested exists, this higher level rigor is used instead.

Each parameter has an associated default value, defined in the `commons.py` file. If your run does not make use of every parameter, it may be desirable to remove those parameter definitions from the parameter file, leaving a more comprehensible file, only stating the definitions you actually care about. An example would be to remove the entire graphics category, if your run does not output any renders or terminal renders. It is however not advised to use these default values in situations where your run depend upon which value is given to some parameter.

6 Snapshot Formats

At the time of writing, two snapshot formats are supported by CONCEPT. One of them is simply referred to as the "standard" CONCEPT snapshot format, while the other is the GADGET-2 format of the second kind*. When using a GADGET-2 snapshot with CONCEPT, certain restrictions apply:

- In the jargon of GADGET-2, dark matter particles are referred to as "halo" particles or particles of type 1. As dark matter is the only implemented species in CONCEPT, for a GADGET-2 snapshot to be usable by CONCEPT, it must consist exclusively of particles of type 1.
- GADGET-2 supports distributing a single snapshot across multiple files. This is not implemented in CONCEPT, and so for a GADGET-2 snapshot to be readable by CONCEPT, the snapshot must be completely contained within a single file.

The reason for making CONCEPT at least somewhat compatible with foreign snapshot types is to connect it with existing software, where the GADGET-2 snapshot format are somewhat standard. The standard CONCEPT snapshot format is however preferable, as it uses the well-established HDF5 format†.

6.1 The Standard CONCEPT Snapshot Format

CONCEPT's own snapshot format, the "standard" format, is really just an HDF5 file with some particular layout. This layout is shown in figure 6.1. The root group contains all of the cosmological parameters also present in parameter files‡. This allows CONCEPT to do sanity checks when reading in an IC file.

Besides attributes containing information about the cosmology, the root group also contain the three `unit`s attributes. Their values are strings telling which physical unit is used as the base unit, for length, time and mass. When a

*For details of the GADGET-2 format of the second kind, see http://wwwmpa.mpa-garching.mpg.de/gadget/users-guide.pdf chapter 6.

†As GADGET-2 *is* able to read and write snapshots in the HDF5 format, this is not a critique of GADGET-2 itself.

‡Here, the scale factor is simply called `a` rather than `a_begin`, as the snapshot is not necessarily intended as an IC file.

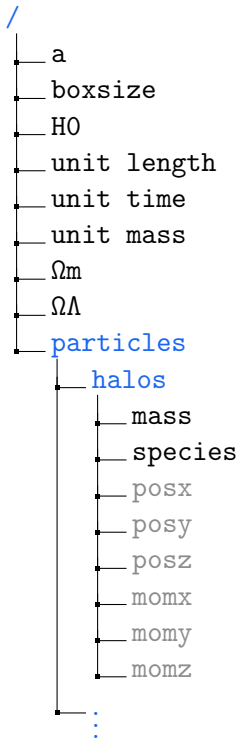

Figure 6.1 – Layout of the standard HDF5 snapshot format used by CONCEPT. The colors indicate groups (blue), attributes (black) and datasets (gray).

standard snapshot is read in by CONCEPT, all physical quantities gets multiplied by the appropriate unit. With the unit information contained in the snapshot, the internal base unit of the current run may differ from that which produced the snapshot. This makes the snapshots more general and self-documenting.

Even with only a single implemented particle species, one may wish to use several groups of particles, which differ in e.g. mass. At the time of writing, the CONCEPT code is not capable of reading in snapshots containing different particles, but the structure of the snapshots allows it. Each group of particles gets a distinctive label called its *type*. In figure 6.1, the only group within the root group is `particles`, which in turn stores all particle types as subgroups. In figure 6.1, `halo` is shown as an example of a particle type. This group contains a `mass` attribute, giving the mass of each particle of the halo type in units of `unit mass`, together with a `species` attribute, which currently must equal `'dark matter'`. The particle positions and momenta are stored as 6 datasets; `posx`, `posy`, `posz` for the position components and `momx`, `momy`, `momz` for the momenta components. Note that since these datasets know their own size, the number of particles is not saved as an attribute. The positions are given in units of `units length`, while the momenta are given in units of

(`unit length`) $\times$ (`unit mass`)/(`unit time`). Naturally, these are the comoving positions and momenta.

7 Utilities

The content of `concept/utilities` is missing from figure 4.1. This is instead shown in figure 7.1. Each Bash script is its own utility, capable of performing a small task. These should be called via the `concept` script as described in section 4.2, i.e. via the `-u` option. Here follows a description of each utility.

`powerspec`
: This utility uses CONCEPT to compute power spectra of particles in snapshots. Invoke it as

```
./concept -u powerspec PATH
```

  where `PATH` is a path to either a snapshot or a directory containing (maybe among other files) at least one snapshot. All the usual options of the `concept` script still apply. If no parameter file is specified, `powerspec.params` is used.

`render`
: This utility uses CONCEPT to produce renders of particles in snapshots. Invoke it as

```
./concept -u render PATH
```

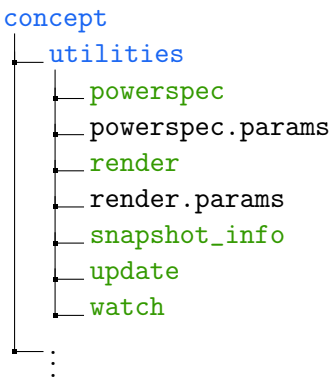

Figure 7.1 – Directory tree focusing on the `utilities` directory. The color codes are identical to those of figure 4.1.

where `PATH` is a path to either a snapshot or a directory containing (maybe among other files) at least one snapshot. All the usual options of the `concept` script still apply. If no parameter file is specified, `render.params` is used.

`snapshot_info`
: This utility will print out the information contained within snapshots. Invoke it as

```
./concept -u snapshot_info PATH
```

where `PATH` is a path to either a snapshot or a directory containing (maybe among other files) at least one snapshot. This utility will always be run locally in pure Python mode, using a single CPU. For standard snapshots, it will simply print out the values of the attributes described in section 6.1, along with the number of particles and their type. The same information is stored in GADGET-2 snapshots, but in very different units. When using the `snapshot_info` utility on a GADGET-2 snapshot, its attribute values will be converted to the "standard" form before being printed out. Comparing a standard snapshot to a GADGET-2 is thus made easy, as all the unit conversions are being taken care of. A GADGET-2 snapshot also contains a lot of other, GADGET-2 specific parameters. These will also get printed out, in their raw form.

`update`
: This utility will update all of the CONCEPT source files to their latest versions, including the `update` script itself. Invoke it either as

```
./concept -u update
```

or directly as

```
utilities/update
```

Needless to say, this utility will always be run locally using a single CPU. Before actually doing the update, a backup of the original CONCEPT code will be placed in `../concept_backup`.

`watch`
: This script will continuously print out the output of a CONCEPT run, after it has been submitted as a PBS job. It does this by reading the produced logfile. If you have multiple jobs submitted, the one submitted most recently will be chosen. Alternatively, the PBS job ID can be passed as an argument when calling the script. That is, call this script either as

```
./concept -u watch [ID]
```

or directly as

```
utilities/watch [ID]
```

Needless to say, this utility will always be run locally using a single CPU. After CONCEPT auto-submits a PBS job, this script is automatically called.

8 Tests

Code validation is achieved trough unit tests. Each subdirectory within the `tests` constitutes such a test, the name of which is equal to the directory name. To run e.g. the `basic` test, invoke `concept` like this:

```
./concept -t basic
```

Each test is controlled by a Bash script named `run_test`, which (with the help of other files in the same directory) runs the CONCEPT code with a certain set of parameters, analyses the output and report any problems. Often, suitable initial conditions are generated for the test, which are then evolved by both CONCEPT and GADGET-2, after which the produced snapshots are compared. Besides subdirectories containing tests, the `tests` directory contains a Bash script called `environment` which handles the setup and teardown of the test environment for all tests.

If a test produces files when run, it should contain a Bash script named `clean`, which when executed deletes these files. The `Makefile` will then execute this script every time you call the `clean_tests` target.

All tests are run locally, meaning without the use of PBS, even when working remotely. The list of available tests is as follows.

`basic`
: Performs a basic test of the CONCEPT environment, making sure that all of the software is correctly build and linked. It simply runs the code without any parameter file in both compiled and pure Python mode, using 1, 2 and 4 processes.

`concept_vs_gadget_P3M`
: Performs a comparison test between CONCEPT's P^3M and GADGET-2's TreePM implementation. Only a single process is used.

`concept_vs_gadget_PM`
: Performs a comparison test between CONCEPT's and GADGET-2's PM implementations. Since GADGET-2 do not expose its pure PM method to the user, edited versions of the GADGET-2 source files `pm_periodic.c` and `timestep.c` are used. Only a single process is used.

`concept_vs_gadget_PP`
: Performs a comparison test between CONCEPT's PP and GADGET-2's tree implementation. Only a single process is used.

`drift`
: Performs a test of CONCEPT's drift operation, by comparing it to that of GADGET-2. ICs containing a particle configuration with no net forces and equal initial velocities are constructed. The particle trajectories are now independent of gravity, making it possible to test the comoving equations of motions. Since gravity is not simply turned off, but rather delicately balanced to produce no net effect, its periodicity is also tested during this test. To test periodicity further, the particles will go trough the side of the box during the simulation. The PP method is used for gravity, and only a single process is used.

`gadget`
: As GADGET-2 is used by many of the other tests, it is important that GADGET-2 itself works properly. This test is simply a shortened version of GADGET-2's own `lcdm_gas` test.

`kick_PP_with_Ewald`
: Performs a test of CONCEPT's PP implementation, by comparing it to GADGET-2's tree implementation. ICs containing a particle configuration with net forces only in the x-direction are constructed. The particles start out with zero velocities, making the study of the kick operation in isolation easy. Any errors in the implementation of the Ewald method should be detected.

`kick_PP_without_Ewald`
: Performs a test of CONCEPT's PP implementation with Ewald summation disabled, by comparing it to GADGET-2's tree implementation. GADGET-2 do not provide a switch to turn off the Ewald summation. To achieve this, the GADGET-2 source file `forcetree.c` is edited accordingly. ICs containing a particle configuration with net forces only in the x-direction are constructed. The particles start out with zero velocities, making the study of the kick operation in isolation easy. Any simple errors in the PP algorithm, having to do with units and the like, should be detected.

`nprocs_P3M`
: Performs a test of the $\mathrm{P}^3\mathrm{M}$ implementation, comparing runs using different numbers of processes. Specifically, 1, 2 4 and 8 processes will be used.

`nprocs_PM`
: Performs a test of the PM implementation, comparing runs using different numbers of processes. Specifically, 1, 2, 4 and 8 processes will be used.

`nprocs_PP`
: Performs a test of the PP implementation, comparing runs using different numbers of processes. Specifically, 1, 2, 4 and 8 processes will be used.

`powerspec`
: Performs a test of the power spectrum computation by comparing σ_8 from this computation with an estimate obtained by taking the rms of the number of particles within cubes the size of spheres with radius 8 Mpc.

`pure_python_P3M`
: Performs a test of the P^3M implementation, comparing a compiled run on a single process to pure Python runs with different numbers of processes. Specifically, 1, 2, and 4 processes will be used. Deviations between the compiled run and the pure Python runs signals an error in the translation from Python to C.

`pure_python_PM`
: Performs a test of the PM implementation, comparing a compiled run on a single process to pure Python runs with different numbers of processes. Specifically, 1, 2, and 4 processes will be used. Deviations between the compiled run and the pure Python runs signals an error in the translation from Python to C.

`pure_python_PP`
: Performs a test of the PP implementation, comparing a compiled run on a single process to pure Python runs with different numbers of processes. Specifically, 1, 2, and 4 processes will be used. Deviations between the compiled run and the pure Python runs signals an error in the translation from Python to C.

`render`
: Generates a random snapshot and renders it first using 1 process and giving the render utility the exact path to the snapshot (with and without specifying a render parameter file). Two copies of this snapshot is then placed in a separate directory. Using 2 processes, the render utility is then given the path to this directory, which should produce a render for each snapshot. Different render parameters are used for the two calls to the render utility. The test will fail if the two renders of the identical snapshots are not themselves identical. The image resolution is also checked. Finally, it is checked whether the text stating the scale factor is clearly visible on both dark and bright backgrounds.

All of the available tests can be run one after another, by

```
./concept -t all
```

This is done as the last step of the CONCEPT installation when using the `installer`, unless the `--fast` option is given.