PC Documentation

Illa R. Losada

Sep 09, 2023

INTRODUCTION

Welcome!

This is the new homepage of The Pencil Code documentation!

Explore the page hierarchy below (or in the sidebar), and get started with *contributing your own documentation*!

The Pencil Code is primarily designed to deal with weakly compressible turbulent flows, which is why we use high-order first and second derivatives. To achieve good parallelization, we use explicit (as opposed to compact) finite differences. Typical scientific targets include driven MHD turbulence in a periodic box, convection in a slab with non-periodic upper and lower boundaries, a convective star embedded in a fully nonperiodic box, accretion disc turbulence in the shearing sheet approximation, self-gravity, non-local radiation transfer, dust particle evolution with feedback on the gas, etc. A range of artificial viscosity and diffusion schemes can be invoked to deal with supersonic flows. For direct simulations regular viscosity and diffusion is being used.

Please find more details on our website.

CHAPTER

ONE

GETTING STARTED

Welcome to The Pencil Code documentation page! To help you get started, here are a few tips for using and contributing to this space:

1.1 Using the documentation

The purpose of this space is to bring together The Pencil Code documentation in a unified format and organized in a logical, hierarchical structure. The ultimate goal is to make the process of locating and reading documentation as easy and enjoyable as possible.

1.2 Contributing to the documentation

In order to contribute to the documentation, you will need to clone the GitHub repository containing the source of the documentation, edit the necessary files, and then push your changes back to the repository. The documentation will then be built and deployed automatically using the ReadTheDocs platform.

The following instructions are for Ubuntu systems (64-bit, 16.04+):

1.2.1 Cloning the repository

Go to a directory and type in a terminal:

```
git clone git@github.com:pencil-code/pencil-code.git
```

If you have a github username 'MY_GITHUB_NAME' and like to submit changes you can use:

```
git clone http://MY_GITHUB_NAME@github.com/pencil-code/pencil-code.git
git config --global credential.helper 'cache --timeout=3600'
git config --global branch.autosetuprebase always
```

After cloning the repository, you can access all the documentation files in the directory:

```
cd pencil-code/doc/readthedocs
```

1.2.2 How to build locally (fast)

Follow these instructions to build the reStructuredText documentation (i.e., manuals), but not the auto-generated code documentation. The build is very fast (few seconds).

1. Make sure you have `sphinx` installed and that `sphinx-build` is in your `PATH`.
2. Make sure you have the following `python3` scripts installed (e.g., with `pip3`):

```
sphinx-rtd-theme
sphinxcontrib-images
sphinx-fortran
sphinx-git
```

3. Build:

```
# Fast: do not build auto-generated code documentation
make fast
```

The html files will be built into *_build/html*.

1.2.3 How to build locally (slow)

Follow these instructions to build the complete documentation, including the auto-generated code documentation. The build is slow (several minutes).

Warning: Sphinx imports the entire *Pencil* package in order to generate the documentation (right now, only the *Pencil Python* module. More to come!)

1. Make sure you have `sphinx` installed and that `sphinx-build` is in your `PATH`.
2. Make sure you have the following `python3` scripts installed (e.g., with `pip3`):

```
sphinx-rtd-theme
sphinxcontrib-images
sphinx-js

astropy
numpy
scipy
```

3. Update your local copy of the repository (in order to have freshly autogenerated documentation) and build:

```
git pull --rebase
```

4. Build:

```
# Slow: build auto-generated code documentation
make html
```

The html files will be built into *_build/html*.

1.2.4 Tips for the Python documentation (numpy style)

The *Pencil Python* documentation follows the numpy style docstring convention.

For a thorough example please see the napoleon extension website.

Tip: To make sure sphinx will be successful in generating the documentation, go to the python directory

```
# from the directory containing conf.py
cd ../../python
python
```

and try to import the `pencil` package. If the import succeeds, it is likely that sphinx will also succeed.

1.2.5 Tips for the IDL documentation

Not yet available.

1.2.6 Tips for the Fortran documentation

Not yet available.

Revision history

- working on the documentation by *Illa* at *2021-08-12 23:45:44*, *432ec0b*
- correcting dependencies by *Illa* at *2021-08-09 20:53:28*, *f316a19*
- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*

CHAPTER

TWO

USING RESTRUCTUREDTEXT

All documentation available on this page is written in the reStructuredText (reST) markup language.

2.1 About reST

reST is a simple markup language for plain text files. It is used to semantically mark parts of a document (e.g., section headings, bold text, lists, source code, etc.) for further processing and uniform rendering.

Sphinx is a documentation generator that, in our case, achieves two goals:

- processes the reST documents and renders them to HTML and PDF;
- autogenerates the code documentation for Python, IDL and Fortran projects (i.e., it extracts and formats lists of classes, functions, etc., each with descriptions based on comments found in the source code).

While the reST (and Sphinx) official documentation pages are exhaustive, they are perhaps not recommended for a beginner, as they necessarily contain a lot of information that is not relevant for our documentation page. We suggest starting with https://rest-sphinx-memo.readthedocs.io/en/latest/ReST.html, which is a quick reference for reST and Sphinx that was specifically created to cover a small subset of features that are likely to be used on a daily basis.

2.2 Converting existing documents

The utility *pandoc* can be used to convert a variety of formats, including Microsoft Word (*doc*, *docx*), Libre Office (*odt*), LaTeX, HTML, XML, and if all else fails even PDF, to reST.

The syntax of the command is:

```
pandoc input.doc -o output.rst
```

where *input.doc* is your input document, in any format other than *rst*.

2.3 Style guideline

2.3.1 Headings

In reST, headings are marked by underlining them with the same character:

```
This is a heading
=================
```

In the The Pencil Code, the following markers should be used, in this order:

```
Title of your page
==================

Section
-------

Subsection
~~~~~~~~~~

Sub-subsection
++++++++++++++
```

You should not use further levels of headings, as it would prevent optimal rendering of the table of contents in the left-hand sidebar. You can structure your document further by using the `.. rubric::` directive.

Do not use `###` and `***`, as they are already used for higher-level headings (e.g., on the main landing page).

2.3.2 Admonitions

The use of admonition directives can greatly enhance the user experience by presenting tips, warnings, important notes, etc. in a way that stands out from the rest of the document.

The following admonition directives are available for the Pencil Code: *attention*, *caution*, *danger*, *error*, *hint*, *important*, *note*, *tip*, *todo*, *warning*.

Any of the previous values can be used as follows:

```
.. note::

   This is a note.
```

producing the following output:

Note: This is a note.

Keep in mind that overuse of admonitions will detract from the document flow too much, and consequently worsen the user experience. **Use them sparingly.**

2.3.3 Images

Three different directives allow for the addition images in the documentation. Please, see this guide for a full description.

1. The simplest one is the `image` directive:

```
.. image:: pics/myimage.png
```

   Accepted options for the directive are the width and alternative text for screen readers:

```
.. image:: pics/myimage.png
   :width: 400
   :height: 100px
   :scale: 50 %
   :alt: alternate text
   :align: right
```

2. The `figure` directive supports all the options of the `image` directive and allows for adding a caption to the figure:

```
.. figure:: pics/myimage.png
   :scale: 50 %
   :alt: Flow patterns in the Sun

   This is the caption of the figure (a simple paragraph).

   This is the legend of the figure, which can include a table:

   +-----------------------+-----------------------+
   | Symbol                | Meaning               |
   +=======================+=======================+
   | .. image:: arrow.png  | Magnetic field lines  |
   +-----------------------+-----------------------+
   | .. image:: lines.png  | Velocity lines        |
   +-----------------------+-----------------------+
```

   There must be blank lines before the caption paragraph and before the legend. To specify a legend without a caption, use an empty comment (“..”) in place of the caption.

3. The `thumbnail` directive allows you expand the image by clicking on it:

```
.. thumbnail:: pics/myimage.png
   :width: 500px
```

2.3.4 Videos

You can add short movies to your documentation by using the `.. video::` directive. Any video that works inside an HTML5 *video* tag can be used (i.e., mp4, webm, ogg). Follow these steps to add your video:

- Add the `.. video::` <video_url> directive in your rst file, where you want the video to be rendered.
- It is not necessary to specify any options (height, width, etc.), but if you want to have a look at the documentation of the extension: https://github.com/sphinx-contrib/video

This is the recommended way of adding videos, since they should not be committed to the *ingdoc* git repository, but rather stored on a separate server.

However, if you absolutely need to store the video with the documentation, follow these steps instead:

- Copy the video file to the directory `_static`. This is necessary at the moment, since we have not found a way (yet) for Sphinx to deploy the file otherwise.
- Add the `.. video::  <relative_path_to_video>` directive in your rst file, where you want the video to be rendered. The path is relative to your rst file, so it will probably look similar to `../_static/video.mp4`.

Revision history

- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*

CHAPTER

THREE

THE PENCIL CODE LINKS

These are the essential project links:

- Project homepage: http://pencil-code.nordita.org/
- Code repository: https://github.com/pencil-code/pencil-code
- Wiki: https://github.com/pencil-code/pencil-code/wiki

Revision history

- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*

CHAPTER

FOUR

DISCUSSION GROUPS

The best way to keep up-to-date with the code updates is to join any of this discussion/work groups:

- Pencil code commits: https://groups.google.com/u/1/g/pencil-code-commits
- Pencil code discuss: https://groups.google.com/u/1/g/pencil-code-discuss
- Python for pencil: https://groups.google.com/u/1/g/pencil-code-python
- Documentation for the Pencil Code: https://groups.google.com/g/pencil-code-doc

---

Revision history

- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*

CHAPTER

FIVE

QUICK START GUIDE

5.1 Required software

5.1.1 Linux

A Fortran and a C compiler are needed to compile the code. Both compilers should belong to the same distribution package and version (e.g. GNU GCC or Intel).

5.1.2 MacOS X

For Mac, you first need to install Xcode from the website http://developer.apple.com/, where you have to register as a member. Alternatively, an easy to install `gfortran` binary package can be found at the website http://gcc.gnu.org/wiki/GFortranBinaries. Just download the archive and use the installer contained therein. It installs into '`/usr/local/gfortran`' with a symbolic link in '`/usr/local/bin/gfortran`'. It might be necessary to add the following line to the "`.cshrc`"-file in your '`/home`' folder:

```
setenv PATH /usr/local/bin:\$PATH
```

5.2 Download the Pencil Code

The Pencil Code is an open source code written mainly in Fortran and available under GPL. General information can be found at our official homepage:

http://pencil-code.nordita.org/.

The latest version of the code can be downloaded with `svn`. In the directory where you want to put the code, type:

```
svn checkout https://github.com/pencil-code/pencil-code/trunk/ pencil-code
```

Alternatively, you may also use `git`:

```
git clone https://github.com/pencil-code/pencil-code.git
```

More details on download options can be found here: http://pencil-code.nordita.org/download.php

The downloaded '`pencil-code`' directory contains several sub-directories:

1. '`doc`': you may build the latest manual as PDF by issuing the command `make` inside this directory
2. '`samples`': contains many sample problems

3. 'config': has all the configuration files
4. 'src': the actual source code
5. 'bin' and 'lib': supplemental scripts
6. 'idl', 'python', 'julia', etc.: data processing for diverse languages

5.3 Configure the shell environment

You need to load some environment variables into your shell. Please change to the freshly downloaded directory:

```
cd pencil-code
```

Probably you use a sh-compatible shell (like the Linux default shell bash), there you just type:

```
. sourceme.sh
```

(In a csh-compatible shell, like tcsh, use this alternative: source sourceme.csh)

5.4 Your first simulation run

5.4.1 Create a new run-directory

Now create a run-directory and clone the input and configuration files from one of the samples that fits you best to get started quickly (here from 'pencil-code/samples/1d-tests/jeans-x'):

```
mkdir -p /data/myuser/myrun/src
cd /data/myuser/myrun
cp $PENCIL_HOME/samples/1d-tests/jeans-x/*.in ./
cp $PENCIL_HOME/samples/1d-tests/jeans-x/src/*.local src/
```

Your run should be put outside of your '/home' directory, if you expect to generate a lot of data and you have a tight storage quota in your '/home'.

5.4.2 Linking to the sources

One command sets up all needed symbolic links to the original Pencil Code directory:

```
pc_setupsrc
```

5.4.3 Makefile and parameters

Two basic configuration files define a simulation setup: "src/Makefile.local" contains a list of modules that are being used, and "src/cparam.local" defines the grid size and the number of processors to be used. Take a quick look at these files…

Single-processor

An example "src/Makefile.local" using the module for only one processor would look like:

```
MPICOMM=nompicomm
```

For most modules there is also a "no"-variant which switches that functionality off.

In "src/cparam.local" the number of processors needs to be set to 1 accordingly:

```
integer, parameter :: ncpus=1,nprocx=1,nprocy=1,nprocz=ncpus/(nprocx*nprocy)
integer, parameter :: nxgrid=128,nygrid=1,nzgrid=128
```

Multi-processor

If you like to use MPI for multi-processor simulations, be sure that you have a MPI library installed and change "src/Makefile.local" to use MPI:

```
MPICOMM=mpicomm
```

Change the ncpus setting in "src/cparam.local". Think about how you want to distribute the volume on the processors — usually, you should have 128 grid points in the x-direction to take advantage of the SIMD processor unit. For compilation, you have to use a configuration file that includes the "_MPI" suffix, see below.

5.4.4 Compiling...

In order to compile the code, you can use a pre-defined configuration file corresponding to your compiler package. E.g. the default compilers are gfortran together with gcc and the code is being built with default options (not using MPI) by issuing the command:

```
pc_build
```

Alternatively, for multi-processor runs (still using the default GNU-GCC compilers):

```
pc_build -f GNU-GCC_MPI
```

Using a different compiler (optional)

If you prefer to use a different compiler package (e.g. with MPI support or using ifort), you may try:

```
pc_build -f Intel
pc_build -f Intel_MPI
pc_build -f Cray
pc_build -f Cray_MPI
```

More pre-defined configurations are found in the directory "pencil-code/config/compilers/*.conf".

Changing compiler options (optional)

Of course you can also create a configuration file in any subdirectory of 'pencil-code/config/hosts/'. By default, pc_build looks for a config file that is based on your host-ID, which you may see with the command:

```
pc_build -i
```

You may add your modified configuration with the filename "host-ID.conf", where you can change compiler options according to the Pencil Code manual. A good host configuration example, that you may clone and adapt according to your needs, is "pencil-code/config/hosts/IWF/host-andromeda-GNU_Linux-Linux.conf".

5.4.5 Running...

The initial conditions are set in "start.in" and the parameters for the main simulation run can be found in "run.in". In "print.in" you can choose which quantities are written to the file "data/time_series.dat".

Be sure you have created an empty 'data' directory.

```
mkdir data
```

It is now time to run the code:

```
pc_run
```

If everything worked well, your output should contain the line

```
start.x has completed successfully
```

after initializing everything successfully. It would then start running, printing in the console the quantities specified in "print.in", for instance,

```
---it--------t-------dt------rhom------urms------uxpt-----uypt-----uzpt-----
      0      0.00 4.9E-03 1.000E+00  1.414E+00 2.00E+00 0.00E+00 0.00E+00
     10      0.05 4.9E-03 1.000E+00  1.401E+00 1.98E+00 0.00E+00 0.00E+00
     20      0.10 4.9E-03 1.000E+00  1.361E+00 1.88E+00 0.00E+00 0.00E+00
     .......
```

ending with

```
Simulation finished after        xxxx  time-steps
.....
Wall clock time/timestep/meshpoint [microsec] = ...
```

An empty file called "COMPLETED" will appear in your run directory once the run is finished.

If you work with one of the samples or an identical setup in a new working directory, you can verify the correctness of the results by checking against reference data, delivered with each sample:

```
diff reference.out data/time_series.dat
```

Welcome to the world of Pencil Code!

5.4.6 Troubleshooting...

If compiling fails, please try the following — with or without the optional `_MPI` for MPI runs:

```
pc_build --cleanall
pc_build -f GNU-GCC_MPI
```

If some step still fails, you may report to our mailing list: http://pencil-code.nordita.org/contact.php. In your report, please state the exact point in this quick start guide that fails for you (including the full error message) — and be sure you precisely followed all non-optional instructions from the beginning.

In addition to that, please report your operating system (if not Linux-based) and the shell you use (if not `bash`). Also please give the full output of these commands:

```
bash
cd path/to/your/pencil-code/
source sourceme.sh
echo $PENCIL_HOME
ls -la $PENCIL_HOME/bin
cd samples/1d-tests/jeans-x/
gcc --version
gfortran --version
pc_build --cleanall
pc_build -d
```

If you plan to use MPI, please also provide the full output of:

```
mpicc --version
mpif90 --version
mpiexec --version
```

5.5 Data post-processing

5.5.1 IDL visualization (optional,)

GUI-based visualization (recommended for quick inspection)

The most simple approach to visualize a Cartesian grid setup is to run the Pencil Code GUI and to select the files and physical quantities you want to see:

```
IDL> .r pc_gui
```

If you miss some physical quantities, you might want to extend the two IDL routines `pc_get_quantity` and `pc_check_quantities`. Anything implemented there will be available in the GUI, too.

Command-line based processing of "big data"

Please check the documentation inside these files:

pencil-code/idl/read/pc_read_var_raw.pro	efficient reading of raw data
pencil-code/idl/read/pc_read_subvol_raw.pro	reading of sub-volumes
pencil-code/idl/read/pc_read_slice_raw.pro	reading of any 2D slice from 3D snapshots
pencil-code/idl/pc_get_quantity.pro	compute physical quantities out of raw data
pencil-code/idl/pc_check_quantities.pro	dependency checking of physical quantities

in order to read data efficiently and compute quantities in physical units.

Command-line based data analysis (may be inefficient)

Several idl-procedures have been written (see in 'pencil-code/idl') to facilitate inspecting the data that can be found in raw format in 'jeans-x/data'. For example, let us inspect the time series data

```
IDL> pc_read_ts, obj=ts
```

The structure ts contains several variables that can be inspected by

```
IDL> help, ts, /structure
** Structure <911fa8>, 4 tags, length=320, data length=320, refs=1:
   IT              LONG      Array[20]
   T               FLOAT     Array[20]
   UMAX            FLOAT     Array[20]
   RHOMAX          FLOAT     Array[20]
```

The diagnostic UMAX, the maximal velocity, is available since it was set in "jeans-x/print.in". Please check the manual for more information about the input files.

We plot now the evolution of UMAX after the initial perturbation that is defined in "start.in":

```
IDL> plot, ts.t, alog(ts.umax)
```

The complete state of the simulation is saved as snapshot files in "jeans-x/data/proc0/VAR*" every dsnap time units, as defined in "jeans-x/run.in". These snapshots, for example "VAR5", can be loaded with:

```
IDL> pc_read_var, obj=ff, varfile="VAR5", /trimall
```

Similarly tag_names will provide us with the available variables:

```
IDL> print, tag_names(ff)
T X Y Z DX DY DZ UU LNRHO POTSELF
```

The logarithm of the density can be inspected by using a GUI:

```
IDL> cslice, ff.lnrho
```

Of course, for scripting one might use any quantity from the ff structure, like calculating the average density:

```
IDL> print, mean(exp(ff.lnrho))
```

5.5.2 Python visualization (optional)

Be advised that the Python support is still not complete or as feature-rich as for IDL. Furthermore, we move to Python3 in 2020, and not all the routines have been updated yet.

Python module requirements

In this example we use the modules: `numpy` and `matplotlib`. A complete list of required module is included in "`pencil-code/python/pencil/README`".

Using the 'pencil' module

After sourcing the "`sourceme.sh`" script (see above), you should be able to import the `pencil` module:

```
import pencil as pc
```

Some useful functions:

`pc.read.ts`	read "`time_series.dat`" file. Parameters are added as members of the class
`pc.read.slices`	read 2D slice files and return two arrays: (nslices,vsize,hsize) and (time)
`pc.visu.animate_interactive`	assemble a 2D animation from a 3D array

Some examples of postprocessing with Python can be found in the :ref:` python documentation <modpython>` and in the :ref:` python tutorials <tutpython>`.

Revision history

- working on the documentation by *Illa* at *2021-08-12 23:45:44*, *432ec0b*
- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*
- adding readthedocs directory by *Illa* at *2021-05-26 02:16:24*, *1c471b0*

CHAPTER

SIX

PENCIL CODE TUTORIALS

The Pencil Code is written in Fortran90, and is hosted at github https://github.com/pencil-code/pencil-code.

6.1 Modifying the Pencil Code

Note: Adapted from this github wiki page

Commit rights for the Pencil Code are given out quite liberally, but they come with responsibility: - only commit code that is meaningful and necessary - avoid committing code that breaks any of the auto tests - discuss major changes with other developers first (e.g. at the pencil-code-discuss mailing list) - follow the existing *Coding Style guide*.

When developing the Pencil Code, we often communicate via commit messages. A typical example is:

```
Fixed a bug in hydro.do_something().
@Paul, you wrote the original subroutine, can you check whether my
fix is OK?
```

and ideally a few commits down the line, we would have

```
Improved Peter's fix of the hydro.do_something() subroutine.
Thanks for finding and analyzing this.
```

For this mode of communication to work, we **must** be able to rely on our co-developers to read the commit messages. If they do not, their contributions have a strongly reduced value for the code, and they probably should not have commit rights in the first place.

By far the easiest way of reading the commit messages is to subscribe to `pencil-code-commits@googlegroups.com` and read at least superficially through the commit messages as they appear in the mailbox. Individual developers may prefer other ways of keeping up to date, but you should be aware that if you do not follow what is going on with the code, it is likely that parts of the code that you contributed will eventually get moved around, altered, or even removed.

If you want to become a member of any of the groups without being a committer, and if your name is not already known to us (or by googling) we would appreciate a brief email to us explaining your interest. This is to prevent spammers entering our lists. The pencil-code-core list, on the other hand, is reserved for project owners only.

6.2 Coding Style guide

Note: Adapted from this github wiki page

We describe here the Pencil Code coding style and best practice to write code in form of a checklist that should applied to each of the given items. Of course, no rules can be hammered in stone and always need some reality check to ensure these rules help improving the code and not the opposite.

6.2.1 Is a module...

- named well in the sense that its name describes its purpose?
- abstract enough to form a separate module?
- consistent within the existing modules scheme?
- interface obviously revealing how the module should be used?
- interface abstract enough so that the module can be used without thinking about how the services are implemented?

6.2.2 Is a subroutine...

- name revealing exactly what this subroutine is doing?
- implementing only one task that is well defined?
- containing only code parts that would not better be put in a separate subroutine?
- interface obviously revealing how the subroutine should be used?
- interface abstract enough so that the subroutine can be used without thinking about how it is implemented in detail?

6.2.3 Is a data type...

- named well so that its name describes its content type?
- descriptive so that it helps to document its variable declaration?
- simple so that it minimizes complexity?
- that needs to be complex operated only through access subroutines (set_XY/get_XY)?

6.2.4 Is a variable...

- really necessary to be variable (and not a constant)?
- given all possible attributes (like "save", "parameter", "intent(in/out)")?
- not some kind of a "magic number" or "magic string" that should be converted to a named constant?
- not redundant and distinct to all other variables that are otherwise available?
- used only for the single purpose that it was intended for by its name?

- additionally defined and used, if the clarity of the code is significantly improved?

6.2.5 Is a variable name...

- chosen well so that it describes its content (i.p. not its data type)?
- readable and following the style "lower_case_words_connected_with_underscores"?
- of a boolean indicating its flag-like behavior (currently "lflag" for a flag)?
- of a loop counter more informative than just i, j, k, l, m, n? ([i,j] should be reserved only for general matrix and vector operations, while [l,m,n] are reserved for the m-n-loop, for handling the f-array or p-pencil indices).

6.2.6 Is a logical statement...

- using only simple boolean expressions?
- better stored it into an additional boolean variable or put into a boolean function, if it is to be reused?
- not using double negations? (oops!)

6.2.7 Is an `if-else`-construct...

- consisting of small blocks where both blocks are of similar size?
- written so that the "normal" case appears first?
- used to minimize complexity?

6.2.8 Is a loop...

- performing exactly one well-defined function, as a subroutine would?
- implemented using the best matching type: `do/while/repeat`?
- nested in another loop only if necessary?

6.2.9 Is the code...

- representing its own logical structure?
- nominal path or calling sequence clear and easy to follow?
- organized so that related statements are grouped together?
- free of relatively independent code blocks that could stay in subroutines?
- hiding implementation details to the greatest extent?
- written in respect to the problem solution and not in terms of programming-language requirements?
- initializing all variables outside any conditional statements? (e.g. code inside `if-elseif-elseif`-constructs might never be executed!)
- compiling without any compiler warnings? (yes, they do have a serious background even if it is sometimes not obvious!)

6.2.10 Is a commit message...

- not "new settings" or "some corrections" or "minor changes" or similar?
- telling which code block is affected?
- containing all relevant major changes?
- informative about the consequences of previous bugs that are now fixed?
- giving hints what to search or where to start reading if anyone is interested in details?

6.2.11 FORTRAN formatting

- use the Fortran95 language standard (F95)
- use spaces for indentation, two spaces represent one indentation level
- use spaces for formatting of tabular data or comments
- no spaces at the end of a line
- one empty line is enough to split code blocks
- two empty lines can be used to split distinct parts of code
- in-code comments should be indented together with the code
- block-like comments (e.g. function headers) start at the beginning of a line
- use spaces around operators, where applicable
- line-breaks are required after 130 characters (F95)
- line-breaks can be used to significantly improve readability of long code lines

6.2.12 Typical rule-breaker and its solution

- `goto` => implement a loop or an `if-else`-construct
- `entry` => implement an interface or split into distinct subroutines
- `format` => put the format string inside each `write` statement
- hard-coded file units => use a named constant
- hard-coded string length => use pre-defined global constants

6.2.13 Recommended further reading

- Kernighan, Brian, and Plauger: "The Elements of Programming Style", 2nd ed., McGraw-Hill, New York, 1978
- Kernighan, Brian, and Pike: "The Practice of Programming", Addison Wesley, Reading (Massachusetts), 1999
- McConnell: "Code Complete", 2nd ed., Microsoft Press, Redmont (Washington), 2004
- Hunt and Thomas: "The Pragmatic Programmer", Addison Wesley, Reading (Massachusetts), 1999

6.3 Social Rules?

Note: Adapted from this github wiki page

Hi guys,

I discussed with a co-developer of a code that is developed pretty much like Pencil, open-source, a team, version control, etc. Talking to him about code development, I asked if there were cases of flame fights or heated arguments in the code community. He mentioned a couple of cases, and pointed me to **books** on open source development where such stuff is discussed. Not surprisingly, it is quite a common occurrence.

Chapter 6 of the first link, from 102 on ("Difficult People"), is particularly relevant.

http://producingoss.com/

http://artofcommunityonline.org/

Wlad.

6.3.1 Good electronic communication

For an electronic discussion, there is no such thing as a meta-level of information transfer. Therefore, every good electronic communicator just stays with the facts. And *if* an interpretation needs to be done, one chooses the interpretation that assumes *best* motives of your opponent. Only then, one has a chance to understand the opponent right. And without understanding an opponent *fully*, one has no right to answer. (Philippe)

6.3.2 Code of Conduct

Although spaces may feel informal at times, we want to remind ourselves that this is a professional space. As such, the Pencil Code community adheres to a code of conduct adapted from the Contributor Covenant (https://www.contributor-covenant.org/) code of conduct. All contributors will be required to confirm they have read our code of conduct, and are expected to adhere to it in all Pencil Code spaces and associated interactions.

Revision history

- Adding the github wiki content to readthedocs and a docstring example for python by *Illa* at *2021-09-21 02:11:21*, *0ea2ac9*

CHAPTER

SEVEN

PENCIL PYTHON TUTORIALS

Here you can find some tutorials on how to modify/contribute to the Python Code using the Coding style *Python Coding Style* and how to use the code for post-processing *Pencil Code Commands in General*.

7.1 Python Coding Style

Good coding style greatly improves the readability of the code. Similar to the guidelines for the Fortran routines, it is strongly recommended to follow some basic style rules for the python routines. These are some recommendations extracted from PEP 008 and Google Python Style Guide.

7.1.1 General Style Guide for Python

Indentation and Spaces

- Use 4 spaces per indentation level.
- Use hanging indent for function calls over multiple lines:

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

- Wildcard imports (from import *) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools.
- More than one space around an assignment (or other) operator to align it with another should be avoided. **No**:

```
x             = 1
y             = 2
long_variable = 3
```

Yes:

```
x = 1
y = 2
long_variable = 3
```

- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+= , -= etc.), comparisons (== , < , > , != , <> , <= , >= , in , not in , is , is not), Booleans (and , or , not).

- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies).

  Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
```

  No:

```
i=i+1
submitted +=1
x = x * 2 - 1
```

- Don't use spaces around the = sign when used to indicate a keyword argument or a default parameter value.

  Yes:

```
def complex(real, imag=0.0):
      return magic(r=real, i=imag)
```

  No:

```
def complex(real, imag = 0.0):
      return magic(r = real, i = imag)
```

Comments

- Comments should be complete sentences.
- Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment). Paragraphs inside a block comment are separated by a line containing a single # .

Docstrings

Always use docstrings for classes and functions which can be accessed by the user.

We are now working with read the docs and sphinx to create automatic documentation for the code, hence we have updated the style guide for creating docstrings.

We are using Numpy docstring style, and require the following fields in the docstring:

- General description of the Class/function
- Signature: how the function can be called
- Parameters: list of parameters of the class/function
- Returns: type of variable the function returns
- Examples: at least one example of usage
- Notes (ptional): any further comments to the function

```
def complex(real=0.0, imag=0.0):
     """
     Form a complex number.

     Signature
     ---------
     complex(real, imag)

     Parameters
     ----------
      *real*: float
          the real part (default 0.0)
      *imag*: float
          the imaginary part (default 0.0)

     Returns
     -------
     complex number with real and imaginary part

     Examples
     --------
     Define two complex numbers:
     >>> a = complex(3, 5)
     >>> b = complex(4, 7)
     >>> print(a)
     (3+5j)
     >>> a + b
     (7+12j)
     """
```

Naming Convention

module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_CONSTANT_NAME, global_var_name, instance_var_name, function_parameter_name, local_var_name

Exceptions for >our< code: datadir, varfile, varfiles, ...

pylint

Run pylint over your code. pylint is a tool for finding bugs and style problems in Python source code. It finds problems that are typically caught by a compiler for less dynamic languages like C and C++.

black

Run black over your code for automatic formatting. This makes sure that all the above criteria (apart from the doc string) are fullfilled.

Default Function Arguments

Do not use mutable objects as default values in the function or method definition.

Yes:

```
def foo(a, b=None):
    if b is None:
        b = []
```

No:

```
def foo(a, b=[]):
```

Private Methods

Python does not know any private methods or class member. In order to somewhat hide such methods use two underscores in the function definition: `def __magicAttributes(self, param):`.

Others

- Use `''.startswith()` and `''.endswith()` instead of string slicing to check for prefixes or suffixes. startswith() and endswith() are cleaner and less error prone. For example: **Yes**: `if foo.startswith('bar'):` **No**: `if foo[:3] == 'bar':`
- For sequences, (strings, lists, tuples), use the fact that empty sequences are false.

  Yes:

  ```
  if not seq:
  if seq:
  ```

  No:

  ```
  if len(seq)
  if not len(seq)
  ```

- Don't compare boolean values to True or False using == .

  Yes:

  ```
  if greeting:
  ```

  No:

```
if greeting == True:
```

- Check if a variable has a particular type by using `isinstance`, e.g.: `isinstance(my_variable, list)`.

7.1.2 Pencil Code Specific Style

Classes/Objects

Use classes as much as possible. When you write a function try to embed it into a class as **init** function which should return the desired result. This has the advantage of adding methods to the returned object which can modify the data. Read-methods always give back objects containing the whole information (container philosophy). Therefore we use classes if possible.

Data Directory

The default data directory is always './data' and not 'data'.

File Headers

Start each file with the file ID and a short description of the routines. (The authors' list is no longer required since it can be easily accesed through git history.)

```
# varfile.py
#
# Read VAR files. Based on the read_var.pro IDL script.
#
# NB: the f array returned is C-ordered: f[nvar,nz,ny,nx]
#     NOT Fortran as in Pencil (& IDL):  f[nx,ny,nz,nvar]
```

Import Libraries

- Import numpy as *np* instead of *N*.
- Import pylab as *plt* instead of *P*.

If you need to access libraries in some routines in your module, import them in the routine, rather than the head of the module. That way they are not visible by the user.

Yes:

```
# my_module.py

class MyClass(object):
    """
    Some documentation.
    """

    def __init__(self):
        import numpy as np

        self.pi = np.pi
```

No:

```
# my_module.py
import numpy as np

class MyClass(object):
"""
Some documentation.
"""

def __init__(self):
        self.pi = np.pi</pre>
```

7.1.3 Further Reading

https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces

https://google-styleguide.googlecode.com/svn/trunk/pyguide.html

7.2 Pencil Code Commands in General

For a list of all Pencil Code commands start IPython and type `pc.` <TAB> (as with auto completion). To access the help of any command just type the command followed by a '?' (no spaces), e.g.:

```
pc.math.dot?
Type:       function
String Form:<function dot at 0x7f9d96cb0cf8>
File:       ~/pencil-code/python/pencil/math/vector_multiplication.py
Definition: pc.math.dot(a, b)
Docstring:
take dot product of two pencil-code vectors a & b with shape

a.shape = (3, mz, my, mx)
```

You can also use `help(pc.math.dot)` for a more complete documentation of the command.

There are various reading routines for the Pencil Code data. All of them return an object with the data. To store the data into a user defined variable type e.g.

```
ts = pc.read.ts()
```

Most commands take some arguments. For most of them there is a default value, e.g.

```
pc.read.ts(file_name='time_series.dat', datadir='data')
```

You can change the values by simply typing e.g.

```
pc.read.ts(datadir='other_run/data')
```

7.3 Reading and Plotting Time Series

Reading the time series file is very easy. Simply type

```
ts = pc.read.ts()
```

and python stores the data in the variable `ts`. The physical quantities are members of the object `ts` and can be accessed accordingly, e.g. `ts.t, ts.emag`. To check which other variables are stored simply do the tab auto completion `ts.<TAB>`.

Plot the data with the matplotlib commands:

```
plt.plot(ts.t, ts.emag)
```

The standard plots are not perfect and need a little polishing. See further down about making pretty plots. You can save the plot into a file using the GUI or with

```
plt.savefig('plot.eps')
```

7.4 Reading and Plotting VAR files and slice files

Read var files:

```
var = pc.read.var()
```

Read slice files:

```
slices = pc.read.slices(field='bb1', extension='xy')
```

This returns an object `slices` with members `t` and `xy`. The last contains the additional member `xy`.

If you want to plot e.g. the x-component of the magnetic field at the central plane simply type:

```
plt.imshow(var.bb[0, 128, :, :].T, origin='lower', extent=[-4, 4, -4, 4], interpolation=
↪'nearest', cmap='hot')
```

For a complete list of arguments of `plt.imshow` refer to its documentation.

For a more interactive function plot use:

```
pc.visu.animate_interactive(slices.xy.bb, slices.t)
```

Warning: arrays from the reading routines are ordered `f[nvar, mz, my, mx]`, i.e. reversed to IDL. This affects reading var files and slice files.

7.5 Create a custom VAR0 or var.dat

With the functionality of writing snapshots directly into `VAR*` or `var.dat` the user can now generate an initial condition directly from a numpy array or modify the last snapshot and continue running. The function to be used is in `python/pencil/io/snapshot.py` and is called `write_snapshot`. Here we outline how to generate an initial condition. For modifying the `var.dat` only the last steps are necessary.

First we need an empty run. For this let us use `samples/kin-dynamo`

```
cd pencil-code/samples/kin-dynamo
pc_setupsrc
```

In principle we can use any initial condition, as we are going to over write it. But it is cleaner to use

```
INITIAL_CONDITION = noinitial_condition
```

in `src/Makefile.local`. Compile and start:

```
make
pc_start
```

This generates a `VAR0` and `var.dat` in every proc directory.

Our snapshot writing routine needs to know the cpu structure. Furthermore, we need to know the indices of the primary variables. The first can be obtained from `src/cparam.local`, while the latter can be read from the newly generated `data/index.pro`. The numpy arrays that are written need to have the shape [nvar, nz, ny, nz] with the correct order of variables and no ghost zones. Optionally, the number of ghost zones, which is usually 3, can be specified.

Putting it all together our python routine would look something like this:

```
import numpy as np
import pencil as pc

# Read the data to obtain the shape of the arrays, rather than the actual data.
var = pc.read.var(trimall=True)

# Modify the data.
var.aa += np.random.random(var.aa.shape)

# Write the new VAR0 and var.dat files.
pc.io.write_snapshot(var.aa, file_name='VAR0', nprocx=1, nprocy=1, nprocz=1)
pc.io.write_snapshot(var.aa, file_name='var.dat', nprocx=1, nprocy=1, nprocz=1)
```

7.6 Examples

Standard plots with any plotting library are not the prettiest ones. The same is true for matplotlib. Here are a few pretty examples of plots where the default style is changed. You can add your commands into a script e.g. `plot_results.py` and execute it from your terminal with `python plot_results.py` or in IPython with `exec(open('plot_results.py').read())`.

The sample we use here is `samples/interlocked-fluxrings`.

Simple plot:

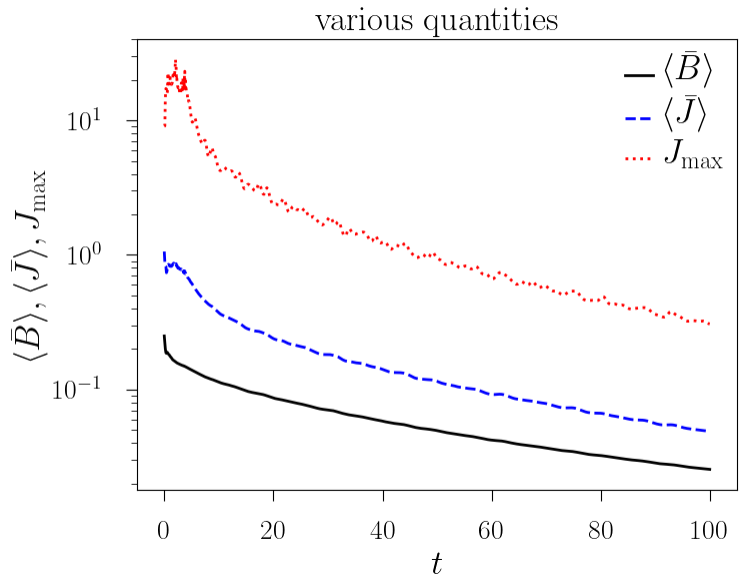
```
import pencil as pc
import numpy as np
import pylab as plt

# Read the time_series.dat.
ts = pc.read.ts()

# Prepare the plot.
# Set the size and margins.
width = 8
height = 6
plt.rc('text', usetex=True)
plt.rc('font', family='arial')
plt.rc("figure.subplot", left=0.2)
plt.rc("figure.subplot", right=0.95)
plt.rc("figure.subplot", bottom=0.15)
plt.rc("figure.subplot", top=0.90)
figure = plt.figure(figsize=(width, height))
axes = plt.subplot(111)

# Make the actual plot.
plt.semilogy(ts.t, ts.brms, linestyle='-', linewidth=2, color='black', label=r'$\langle\
↪bar{B}\rangle$')
plt.semilogy(ts.t, ts.jrms, linestyle='--', linewidth=2, color='blue', label=r'$\langle\
↪bar{J}\rangle$')
plt.semilogy(ts.t, ts.jmax, linestyle=':', linewidth=2, color='red', label=r'$J_{\rm max}
↪$')

plt.xlabel(r'$t$', fontsize=25)
plt.ylabel(r'$\langle\bar{B}\rangle, \langle\bar{J}\rangle, J_{\rm max}$', fontsize=25)
plt.title('various quantities', fontsize=25, family='serif')

# Prepare the legend.
plt.legend(loc=1, shadow=False, fancybox=False, numpoints=1)
leg = plt.gca().get_legend()
# Change the font size of the legend.
ltext = leg.get_texts() # all the text.Text instance in the legend
for k in range(len(ltext)):
        legLine = ltext[k]
        legLine.set_fontsize(25)
frame = leg.get_frame()
frame.set_facecolor('1.0')
leg.draw_frame(False)

# Make plot pretty.
plt.xticks(fontsize=20, family='serif')
plt.yticks(fontsize=20, family='serif')
axes.tick_params(axis='both', which='major', length=8)
axes.tick_params(axis='both', which='minor', length=4)

# Create an offset between the xylabels and the axes.
for label in axes.xaxis.get_ticklabels():
        label.set_position((0, -0.03))
```

(continues on next page)

(continued from previous page)

```
for label in axes.yaxis.get_ticklabels():
        label.set_position((-0.03, 0))
```

The result is this plot:



Simple 2d plot:

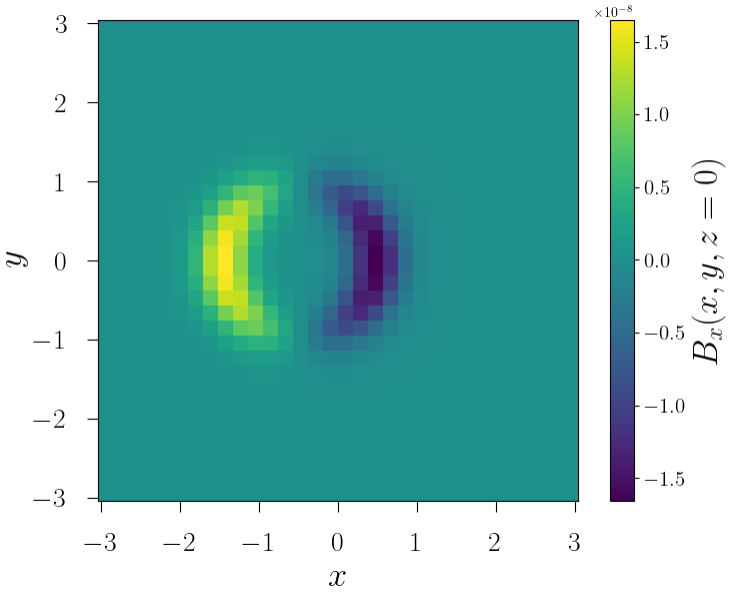
```
import pencil as pc
import numpy as np
import pylab as plt

# Read the slices.
slices = pc.read.slices()

# Read the grid size.
grid = pc.read.grid()
x0 = grid.x[3]
x1 = grid.x[-4]
y0 = grid.y[3]
y1 = grid.y[-4]

# Prepare the plot.
# Set the size and margins.
width = 8
height = 6
plt.rc('text', usetex=True)
plt.rc('font', family='arial')
plt.rc("figure.subplot", left=0.15)
plt.rc("figure.subplot", right=0.95)
plt.rc("figure.subplot", bottom=0.15)
plt.rc("figure.subplot", top=0.95)
figure = plt.figure(figsize=(width, height))
axes = plt.subplot(111)
```

(continues on next page)

(continued from previous page)

```
# Make the actual plot.
plt.imshow(slices.xy.bb1[0, :, :].T, origin='lower', interpolation='nearest',
           extent=[x0, x1, y0, y1])
plt.xlabel(r'$x$', fontsize=25)
plt.ylabel(r'$y$', fontsize=25)

# Set the colorbar.
cb = plt.colorbar()
cb.set_label(r'$B_{x}(x,y,z=0)$', fontsize=25)
cbytick_obj = plt.getp(cb.ax.axes, 'yticklabels')
plt.setp(cbytick_obj, fontsize=15, family='serif')

# Make plot pretty.
plt.xticks(fontsize=20, family='serif')
plt.yticks(fontsize=20, family='serif')
axes.tick_params(axis='both', which='major', length=8)
axes.tick_params(axis='both', which='minor', length=4)

# Create an offset between the xylabels and the axes.
for label in axes.xaxis.get_ticklabels():
        label.set_position((0, -0.03))
for label in axes.yaxis.get_ticklabels():
        label.set_position((-0.03, 0))
```

The result is this plot:



7.7 IDL to Python guide

A large array of idl scripts have been developed over the years, and many of them served their purpose at the time, but there are many others of general purpose. Below is a small selection of examples of idl call sequences along with their python counterparts.

Here are the links to a few potentially useful sites:

1. IDL to Python bridge
2. IDL commands in numerical Python

IDL	Python
pc_read_var,obj=var,/trimall	var = pc.read.var(var_file = 'var.dat', trimall = True, sim = SIM)
help,var	help(var)
pc_read_param,obj=param	pc.read.param()

Revision history

- Added sample plots to python tutorial. by *iomsn* at *2022-02-02 18:31:07*, *13bab5e*
- Corrected a few typos in the Python tutorial for readthedocs. Improved the long examples. by *iomsn* at *2022-02-02 18:12:53*, *b8ed5cb*
- fixing bug in documentation text by *Illa* at *2021-09-21 02:15:31*, *f2fe302*
- Adding the github wiki content to readthedocs and a docstring example for python by *Illa* at *2021-09-21 02:11:21*, *0ea2ac9*
- working on the documentation by *Illa* at *2021-08-12 23:45:44*, *432ec0b*
- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*

CHAPTER

EIGHT

PENCIL MATHEMATICA TUTORIALS

Here you can find some tutorials on using Mathematica for post-processing.

8.1 Loading the package

We need to modify `init.m` so that the path to the package is automatically added to the `$Path` variable in Mathematica. First, type

```
FileNameJoin[{$UserBaseDirectory, "Kernel", "init.m"}]
```

in Mathematica to locate this `init.m` file. Then, add lines

```
AppendTo[$Path, "your/pencil/home/mathematica"]
AppendTo[$Path, "your/pencil/home/mathematica/special"]
```

in this file and save it. In general the path will be `$PENCIL_HOME/mathematica/`, but of course you may put it somewhere else. Mathematica will not search in subdirectories, so make sure the package in right in the folder.

After updating `init.m`, restart the Mathematica kernel (`Evaluation -> Quit Kernel`). To use the package, call `Needs["pc`"]` and then `pcInitialize[]` in a notebook or a script.

To use the package on subkernels, call `pcParallelize[n]`. This will launch `n` subkernels and load the package on each of them. Then you can do things like `ParallelTable[readTS[...],...]`. Only loading the package on the master kernel is not enough. See the discussions here, and the 'Possible issues' section here.

Each time you have updated the data, remember to do `pcInitialize[]` and `pcParallelize[n]` again. These two functions remove some persistent variables defined.

8.2 Pencil Code Commands in General

For a list of all Pencil Code commands, load the package and type `pcFunction[]`. To access the help of any command just type '?' followed by the command, e.g. `?readTS`. You can also check the full definition of the command by typing '??' followed by the command.

8.3 Reading and Plotting Time Series

To read the time series, type

```
data = readTS[sim,var1,var2,...]
```

where `var1`, `var2` etc. are entries in the time series. The return of the right side is a `List` object, with its elements corresponding to `var1`, `var2` etc. You can then access, for example, the time series of `var2` through `data[[2]]` (indexing in Mathematica starts from 1).

Alternatively, you may also put a `List` object on the left side so that `var1`, `var2` etc. will be assigned to each of its elements. For example,

```
{t,urms} = readTS[sim,"t","urms"]
```

Make sure that `Length` of the left side is equal to the number of `var`; otherwise Mathematica will complain.

To plot the data, you can say

```
fig = ListPlot[Transpose[{t,urms}],Joined->True]
```

or, in a one-line command,

```
(** same as ListPlot[Transpose[readTS[sim,"t","urms"]]] **)
fig = readTS[sim,"t","urms"]//Transpose//ListPlot
```

A few options for some internal plotting functions have been reset by the package. For details check `??pcLabelStyle` and `??pcPlotStyle`.

To export the figure in `.eps` format,

```
Export["directory/to/export/figure.eps",fig]
```

8.4 Reading VAR files

VAR files can be read using

```
data = readVARN[sim,iVAR]
```

Here `iVAR` is the index of the VAR file and starts from 0.

By default, ghost zones will not be trimmed. You can do it using the option `"ltrim"->True`, or `"ltrim"->More`; the latter will trim `2*nghost` cells on each boundary.

To compute "magic" variables, you can use

```
data = readVARN[sim,iVAR,{"oo","bb","jj"}]
```

Here "oo", "bb", "jj" refer to vorticity, magnetic, and current fields, respectively.

The return of `readVARN` is an `Association` object (i.e., `Head[data]=Association`). You can obtain all of its keys by `Keys[data]`. Here is an example of its return:

```
data = readVARN[sim,iVAR,{"oo","bb","jj"}];
Keys[data]
(* {"t", "dx", "dy", "dz", "deltay", "lx", "ly", "lz", "x", "y", "z",
   "uu1", "uu2", "uu3", "lnrho", "ooo1", "ooo2", "ooo3", "bbb1", "bbb2",
   "bbb3", "jjj1", "jjj2", "jjj3"} *)
```

Magic variables are named using triple characters, to avoid shadowing the auxilliary ones written by the code (which will be "oo1" etc.).

The `x` coordinates of the mesh points is then `data["x"]`, which will have length `(16+6)^3` if the resolutoin is `16^3` and `nghost=3`. One can form a three-dimensional map of `uu1` using

```
uu1 = Transpose[ data/@{"x","y","z","uu1"} ];
(* {{x1,y1,z1,f1},{x2,y2,z2,f2},...} *)
```

Sometimes the following method is also useful:

```
Clear[uu1]
grid = Transpose[ data/@{"x","y","z"} ];
uu1 = Association[ Thread[ grid->data["uu1"] ] ];
```

Then `uu1` becomes a "function" and its value at `{x1,y1,z1}` is simply `uu1[{x1,y1,z1}]`.

8.5 Visualizing slices from VAR files

A quick way to make a density plot from `data` is

```
showSlice[data, "uu1", {"z", 0.2}]
```

Here `{"z",0.2}` instructs to plot the `xy` slice closest to `z=0.2`.

For vector fields one can also use

```
showSliceVector[data, "uu", {"z", 0.2}]
```

Notice the second argument is just `"uu"` with no index. The function then makes a density plot of the out-of-plane component of (here `"uu3"`), and a superposed vector plot of the in-plane components (here `"uu1"` and `"uu2"`).

8.6 Reading video files

To read video or slice files, one uses

```
{slices,times,position}=readSlice[sim,"uu1","xy2"]
```

The returned `slices` variable is a `List` of all slices at different times, and can be visualized by, say, `DensityPlot[ slices[[1]] ]`. `position` tells you the spatial coordinate of the slices.

Here is an example to make a video:

```
Clear[makeFrame]
makeFrame[ slice_,time_ ] := DensityPlot[ slice, PlotLabel->"t="<>ToString@time]
frames = MapThread[ makeFrame, {slices,times} ];
```

(continues on next page)

(continued from previous page)

```
(* to view the video in the notebook; can be slow if too many frames*)
ListAnimate[ frame, AnimationRunning->False ]
(* output to a movie file *)
Export[ "your/output/directory/video.mov", frames, FrameRate->24 ]
```

One can also visualize variables in a 3D box. For more information see the comments of `makeBox` and `makeBoxes`.

8.7 Running on supercomputers

First, make sure Mathematica is available on the machine. You can check this by saying `which wolfram` in the terminal. If it is not installed, contact your administrator to see if it can be loaded.

Once you have loaded the Mathematica module, try `wolfram` in the terminal. It should bring you to the text-based interface of Mathematica. You can then follow the steps in the previous sections to set up the package.

There is a sample script in the directory `$PENCIL_HOME/mathematica/sample_script.wls`. Modify its first line according to where your `wolfram` is. Remember to include the `-script` option.

To run a script, use `wolframscript your_script.wls`.

Revision history

- introduced the folder mathematica/special, to place packages that are specific to projects by *Hongzhe Zhou* at *2022-06-16 15:55:55*, *0ff0c1b*
- update mathematica tutorial by *Hongzhe Zhou* at *2022-05-03 06:32:56*, *896ed01*
- tutorial on how to run scripts by *Hongzhe Zhou* at *2022-04-25 13:26:45*, *f7f9655*
- update the mathematica package for pcParallelize[] by *Hongzhe Zhou* at *2022-04-12 09:50:59*, *7cec76d*
- update mathematica doc by *Hongzhe Zhou* at *2022-04-07 12:45:29*, *5dc4c00*
- update mathematica tutorial, and minor change to the package by *Hongzhe Zhou* at *2021-12-13 10:36:35*, *648d6f5*
- updates for the mathematica pacakge and tutorial by *Hongzhe Zhou* at *2021-10-14 12:13:22*, *23d67e1*
- mathematica package changed default trimming to False in readVARN; improved some help messages by *Hongzhe Zhou* at *2021-08-17 08:49:06*, *64ccc36*
- added the tutorial page for mathematica package by *Hongzhe Zhou* at *2021-08-16 14:57:31*, *33d91b3*

CHAPTER

NINE

PENCIL PYTHON CODE DOCUMENTATION

9.1 `pencil`: Pencil package

Revision history

- changing documentation configuration and documentation in files by *Illa* at *2021-09-20 18:38:09*, *f582bb0*
- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*

CHAPTER

TEN

IDL CODE DOCUMENTATION

Soon!

Revision history

- working on the documentation by *Illa* at *2021-08-12 23:45:44*, *432ec0b*
- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*

CHAPTER

ELEVEN

FORTRAN MODULE

Testing…

11.1 Module ascalar

Description

Quick access

Variables

a1, *acc_const*, *acc_mean*, *accm_volume*, *amplacc*, *ampl_ttc*, *ascalar_diff*, *ascalar_sink*, *buoyancy*, *const1_qvs*, *const2_qvs*, *consttt*, *cp_constant*, *es_t*, *gradacc0*, *gradtt0*, *gravity_acceleration*, *idiag_acc_mean*, *idiag_accm*, *idiag_accmax*, *idiag_accmin*, *idiag_accrms*, *idiag_buoyancym*, *idiag_buoyancymax*, *idiag_buoyancymin*, *idiag_buoyancyrms*, *idiag_condensationratem*, *idiag_condensationratemax*, *idiag_condensationratemin*, *idiag_condensationraterms*, *idiag_esm*, *idiag_esmax*, *idiag_esmin*, *idiag_esrms*, *idiag_qvsm*, *idiag_qvsmax*, *idiag_qvsmin*, *idiag_qvsrms*, *idiag_ssatm*, *idiag_ssatmax*, *idiag_ssatmin*, *idiag_ssatrms*, *idiag_tauascalarmax*, *idiag_tauascalarmin*, *idiag_tauascalarrms*, *idiag_ttc_mean*, *idiag_ttcm*, *idiag_ttcmax*, *idiag_ttcmin*, *idiag_ttcrms*, *idiag_uxaccm*, *idiag_uyaccm*, *idiag_uzaccm*, *idiag_watermixingratiom*, *idiag_watermixingratiomax*, *idiag_watermixingratiomin*, *idiag_watermixingratiorms*, *initacc*, *initlntt*, *inittt*, *initttc*, *l_t_source*, *lascalar_sink*, *latent_heat*, *lbuoyancy*, *lcondensation_rate*, *lconsttt*, *ltauascalar*, *ltt_mean*, *lttc*, *lttc_mean*, *lupdraft*, *lupw_acc*, *lupw_ttc*, *noascalar*, *qv_env*, *qvs_t*, *rascalar_sink*, *reinitialize_acc*, *rhoa*, *rv*, *rv_over_rd_minus_one*, *ssat0*, *t_env*, *thermal_diff*, *tt_mean*, *ttc_const*, *ttc_mean*, *ttcm_volume*, *updraft*, *vapor_mixing_ratio_qvs*, *widthacc*, *widthttc*

Routines

calc_accmean(), *calc_pencils_ascalar()*, *calc_ttcmean()*, *dacc_dt()*, *init_acc()*, *initialize_ascalar()*, *pencil_criteria_ascalar()*, *pencil_interdep_ascalar()*, *read_ascalar_init_pars()*, *read_ascalar_run_pars()*, *register_ascalar()*, *rprint_ascalar()*, *write_ascalar_init_pars()*, *write_ascalar_run_pars()*

Needed modules

- cdata
- cparam
- messages

Variables

- ascalar/**a1** *[real,private/optional/default=0.0]*
- ascalar/**acc_const** *[real,private/optional/default=0.0]*
- ascalar/**acc_mean** *[real,private/optional/default=0.01]*
- ascalar/**accm_volume** (nx,ny,nz) *[real,private]*
- ascalar/**amplacc** *[real,private/optional/default=0.0]*
- ascalar/**ampllttc** *[real,private/optional/default=0.0]*
- ascalar/**ascalar_diff** *[real,private/optional/default=0.0]*
- ascalar/**ascalar_sink** *[real,private/optional/default=0.0]*
- ascalar/**buoyancy** (nx) *[real,private/optional/default=0.0]*
- ascalar/**const1_qvs** *[real,private/optional/default=253000000000.0]*
- ascalar/**const2_qvs** *[real,private/optional/default=5420.0]*
- ascalar/**consttt** *[real,private/optional/default=293.25]*
- ascalar/**cp_constant** *[real,private/optional/default=1005.0]*
- ascalar/**es_t** (nx) *[real,private/optional/default=0.0]*
- ascalar/**gradacc0** (3) *[real,private/optional/default=(/0.0,0.0,0.0/)]*
- ascalar/**gradtt0** (3) *[real,private/optional/default=(/0.0,0.0,0.0/)]*
- ascalar/**gravity_acceleration** *[real,private/optional/default=9.81]*
- ascalar/**idiag_acc_mean** *[integer,private/optional/default=0]*
- ascalar/**idiag_accm** *[integer,private/optional/default=0]*
- ascalar/**idiag_accmax** *[integer,private/optional/default=0]*
- ascalar/**idiag_accmin** *[integer,private/optional/default=0]*
- ascalar/**idiag_accrms** *[integer,private/optional/default=0]*
- ascalar/**idiag_buoyancym** *[integer,private/optional/default=0]*
- ascalar/**idiag_buoyancymax** *[integer,private/optional/default=0]*
- ascalar/**idiag_buoyancymin** *[integer,private/optional/default=0]*

- ascalar/**idiag_buoyancyrms** *[integer,private/optional/default=0]*
- ascalar/**idiag_condensationratem** *[integer,private/optional/default=0]*
- ascalar/**idiag_condensationratemax** *[integer,private/optional/default=0]*
- ascalar/**idiag_condensationratemin** *[integer,private/optional/default=0]*
- ascalar/**idiag_condensationraterms** *[integer,private/optional/default=0]*
- ascalar/**idiag_esm** *[integer,private/optional/default=0]*
- ascalar/**idiag_esmax** *[integer,private/optional/default=0]*
- ascalar/**idiag_esmin** *[integer,private/optional/default=0]*
- ascalar/**idiag_esrms** *[integer,private/optional/default=0]*
- ascalar/**idiag_qvsm** *[integer,private/optional/default=0]*
- ascalar/**idiag_qvsmax** *[integer,private/optional/default=0]*
- ascalar/**idiag_qvsmin** *[integer,private/optional/default=0]*
- ascalar/**idiag_qvsrms** *[integer,private/optional/default=0]*
- ascalar/**idiag_ssatm** *[integer,private/optional/default=0]*
- ascalar/**idiag_ssatmax** *[integer,private/optional/default=0]*
- ascalar/**idiag_ssatmin** *[integer,private/optional/default=0]*
- ascalar/**idiag_ssatrms** *[integer,private/optional/default=0]*
- ascalar/**idiag_tauascalarmax** *[integer,private/optional/default=0]*
- ascalar/**idiag_tauascalarmin** *[integer,private/optional/default=0]*
- ascalar/**idiag_tauascalarrms** *[integer,private/optional/default=0]*
- ascalar/**idiag_ttc_mean** *[integer,private/optional/default=0]*
- ascalar/**idiag_ttcm** *[integer,private/optional/default=0]*
- ascalar/**idiag_ttcmax** *[integer,private/optional/default=0]*
- ascalar/**idiag_ttcmin** *[integer,private/optional/default=0]*
- ascalar/**idiag_ttcrms** *[integer,private/optional/default=0]*
- ascalar/**idiag_uxaccm** *[integer,private/optional/default=0]*
- ascalar/**idiag_uyaccm** *[integer,private/optional/default=0]*
- ascalar/**idiag_uzaccm** *[integer,private/optional/default=0]*
- ascalar/**idiag_watermixingratiom** *[integer,private/optional/default=0]*
- ascalar/**idiag_watermixingratiomax** *[integer,private/optional/default=0]*
- ascalar/**idiag_watermixingratiomin** *[integer,private/optional/default=0]*

- ascalar/**idiag_watermixingratiorms** *[integer,private/optional/default=0]*
- ascalar/**initacc** *[character,private/optional/default='nothing']*
- ascalar/**initlntt** *[character,private/optional/default='nothing']*
- ascalar/**inittt** *[character,private/optional/default='nothing']*
- ascalar/**initttc** *[character,private/optional/default='nothing']*
- ascalar/**l_t_source** *[logical,private/optional/default=.true.]*
- ascalar/**lascalar_sink** *[logical,private/optional/default=.false.]*
- ascalar/**latent_heat** *[real,private/optional/default=2500000.0]*
- ascalar/**lbuoyancy** *[logical,private/optional/default=.false.]*
- ascalar/**lcondensation_rate** *[logical,private/optional/default=.false.]*
- ascalar/**lconsttt** *[logical,private/optional/default=.false.]*
- ascalar/**ltauascalar** *[logical,private/optional/default=.false.]*
- ascalar/**ltt_mean** *[logical,private/optional/default=.false.]*
- ascalar/**lttc** *[logical,private/optional/default=.false.]*
- ascalar/**lttc_mean** *[logical,private/optional/default=.false.]*
- ascalar/**lupdraft** *[logical,private/optional/default=.false.]*
- ascalar/**lupw_acc** *[logical,private/optional/default=.false.]*
- ascalar/**lupw_ttc** *[logical,private/optional/default=.false.]*
- ascalar/**noascalar** *[logical,private/optional/default=.false.]*
- ascalar/**qv_env** *[real,private/optional/default=0.0163]*
- ascalar/**qvs_t** (nx) *[real,private/optional/default=0.0]*
- ascalar/**rascalar_sink** *[logical,private/optional/default=.false.]*
- ascalar/**reinitialize_acc** *[logical,private/optional/default=.false.]*
- ascalar/**rhoa** *[real,private/optional/default=1.06]*
- ascalar/**rv** *[real,private/optional/default=461.5]*
- ascalar/**rv_over_rd_minus_one** *[real,private/optional/default=0.608]*
- ascalar/**ssat0** *[real,private/target/optional/default=0.0]*
- ascalar/**t_env** *[real,private/optional/default=293.0]*
- ascalar/**thermal_diff** *[real,private/optional/default=0.0]*
- ascalar/**tt_mean** *[real,private/optional/default=293.25]*
- ascalar/**ttc_const** *[real,private/optional/default=0.0]*

- ascalar/**ttc_mean** *[real,private/optional/default=293.0]*
- ascalar/**ttcm_volume** (nx,ny,nz) *[real,private]*
- ascalar/**updraft** *[real,private/optional/default=0.0]*
- ascalar/**vapor_mixing_ratio_qvs** *[real,private/optional/default=0.0]*
- ascalar/**widthacc** *[real,private/optional/default=0.0]*
- ascalar/**widthttc** *[real,private/optional/default=0.0]*

Subroutines and functions

subroutine ascalar/**register_ascalar**()

Initialise the acc variable and increase nvar accordingly

3-jun-16/xiangyu: adapted from pscalar_nolog

Use
farraymanager

Call to
init_acc(), *calc_ttcmean()*, *calc_accmean()*

subroutine ascalar/**initialize_ascalar**(*f*)

Perform any necessary post-parameter read initialization Since the passive scalar is often used for diagnostic purposes one may want to reinitialize it to its initial distribution.

Parameters
f (mx,my,mz,mfarray) *[real]*

Use
sharedvariables (put_shared_variable(), get_shared_variable())

Call to
init_acc(), *calc_ttcmean()*, *calc_accmean()*

subroutine ascalar/**init_acc**(*f*)

initialise passive scalar field; called from start.f90

Parameters
f (mx,my,mz,mfarray) *[real]*

Use
sub, initcond

Called from
register_ascalar(), *initialize_ascalar()*

Call to
calc_ttcmean(), *calc_accmean()*

subroutine ascalar/**pencil_criteria_ascalar**()

All pencils that the Ascalar module depends on are specified here.

Call to
calc_ttcmean(), *calc_accmean()*

subroutine ascalar/**pencil_interdep_ascalar**(*lpencil_in*)

Interdependency among pencils provided by the Pscalar module is specified here.

Parameters
lpencil_in (npencils) *[logical]*

Call to
calc_ttcmean(), *calc_accmean()*

subroutine ascalar/**calc_pencils_ascalar**(*f*, *p*)

Calculate ascalar Pencils. Most basic pencils should come first, as others may depend on them.

Parameters

- **f** (mx,my,mz,mfarray) *[real,in]*
- **p** *[pencil_case,inout]*

Use
sub

Call to
calc_ttcmean(), *calc_accmean()*

subroutine ascalar/**dacc_dt**(*f*, *df*, *p*)

Active scalar evolution for supersation Calculate dacc/dt=-uu.gacc + supersat_diff*[del2acc + glnrho.gacc].

27-may-16/xiangyu: adapted from pscalar_nolog
4-sep-16/axel: added more diagnostics

Parameters

- **f** (mx,my,mz,mfarray) *[real]*
- **df** (mx,my,mz,mvar) *[real,out]*
- **p** *[pencil_case]*

Use
diagnostics, sub

Call to
calc_ttcmean(), *calc_accmean()*

subroutine ascalar/**calc_ttcmean**(*f*)

Calculation of volume averaged mean temperature and water vapor mixing ratio.

06-June-18/Xiang-Yu.Li: coded

Parameters
f (mx,my,mz,mfarray) *[real,in]*

Use
sub (finalize_aver()), diagnostics (save_name())

Called from
register_ascalar(), *initialize_ascalar()*, *init_acc()*, *pencil_criteria_ascalar()*, *pencil_interdep_ascalar()*, *calc_pencils_ascalar()*, *dacc_dt()*

subroutine ascalar/**calc_accmean**(*f*)

Calculation of volume averaged water vapor mixing ratio.

06-June-18/Xiang-Yu.Li: coded

Parameters
f (mx,my,mz,mfarray) *[real,in]*

Use
sub (finalize_aver()), diagnostics (save_name())

Called from
register_ascalar(), *initialize_ascalar()*, *init_acc()*, *pencil_criteria_ascalar()*, *pencil_interdep_ascalar()*, *calc_pencils_ascalar()*, *dacc_dt()*

subroutine ascalar/**read_ascalar_init_pars**(*iostat*)

Parameters
iostat *[integer,out]*

Use
file_io (parallel_unit())

subroutine ascalar/**write_ascalar_init_pars**(*unit*)

Parameters
unit *[integer,in]*

subroutine ascalar/**read_ascalar_run_pars**(*iostat*)

Parameters
iostat *[integer,out]*

Use
file_io (parallel_unit())

subroutine ascalar/**write_ascalar_run_pars**(*unit*)

Parameters
unit *[integer,in]*

subroutine ascalar/**rprint_ascalar**(*lreset*[, *lwrite*])

Parameters

- **lreset** *[logical]*
- **lwrite** *[logical]*

Use
diagnostics, farraymanager (farray_index_append())

11.2 Module Geometrical Types

Description

Id

MODULE_DOC: Collection of geometrical object types. MODULE_DOC: (Presently only rectangular toroid)

16-May-20/MR: coded

Quick access

Types
torus_rect

Variables
torus_extend_r, *torus_extend_z*, *torus_init*, *torus_precess*, *torus_rect*, *torus_rect_unfmt_read*, *torus_rect_unfmt_write*, *torus_wobble*

Types

- **type** geometrical_types/**torus_rect**

  Type fields

  - **% center** (3) *[real]*
  - **% center0** (3) *[real]*
  - **% extr_rate** *[real]*
  - **% extz_rate** *[real]*
  - **% height** *[real,optional/default=0.0]*
  - **% height0** *[real]*
  - **% omega_prec** *[real]*
  - **% ph** *[real,optional/default=0.0]*
  - **% ph0** *[real]*
  - **% r_in** *[real,optional/default=0.0]*
  - **% r_in0** *[real]*
  - **% th** *[real,optional/default=0.0]*
  - **% th0** *[real]*
  - **% thick** *[real,optional/default=0.0]*
  - **% wob_amp** (3) *[real]*
  - **% wob_om** (3) *[real]*
  - **% wob_phase** (3) *[real]*

Variables

- geometrical_types/**torus_extend_r** *[public]*
- geometrical_types/**torus_extend_z** *[public]*
- geometrical_types/**torus_init** *[public]*
- geometrical_types/**torus_precess** *[public]*
- geometrical_types/**torus_rect** *[public]*
- geometrical_types/**torus_rect_unfmt_read** *[public]*
- geometrical_types/**torus_rect_unfmt_write** *[public]*
- geometrical_types/**torus_wobble** *[public]*

Revision history

- changing documentation configuration and documentation in files by *Illa* at *2021-09-20 18:38:09*, *f582bb0*
- working on the documentation by *Illa* at *2021-08-12 23:45:44*, *432ec0b*
- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*
- adding readthedocs directory by *Illa* at *2021-05-26 02:16:24*, *1c471b0*

Revision history

- Adding the github wiki content to readthedocs and a docstring example for python by *Illa* at *2021-09-21 02:11:21*, *0ea2ac9*
- added the tutorial page for mathematica package by *Hongzhe Zhou* at *2021-08-16 14:57:31*, *33d91b3*
- working on the documentation by *Illa* at *2021-08-12 23:45:44*, *432ec0b*
- Updating all the readthedocs documentation by *Illa* at *2021-08-09 20:45:37*, *3901ba4*
- adding readthedocs directory by *Illa* at *2021-05-26 02:16:24*, *1c471b0*

FORTRAN MODULE INDEX

INDEX

L

N

P

Q

R

S

T

U

V

W