Introduction to `mn-fysrp-pic`

Sigvald Marholm

12.10.15

1 Introduction

The `mn-fysrp-pic` Git repository holds the official Particle-In-Cell (PIC) code belonging to the 4Dspace project and the Plasma and Space Physics group at the Physics Department of UiO. In order to keep the PIC code and their different versions clean and manageable and to avoid conflicts during cooperation it is of utmost importance that all users obey the rules of the repository. Each user is therefore responsible of making himself/herself familiar with the rules stated herein. Failure to do so may result in reduced privileges in the repository.

2 Workflow

The `mn-fysrp-pic` repository utilizes what's called a Feature Branch Workflow[1] as illustrated in Fig. 1. The reason for this is to allow several users to develop functions for the code independently with a minimum of conflicts. It also assures that a fully functioning version of the code is always accessible. Using Git also means that all previous revisions of the code are retrievable. This, along with information on which revision was used to generate a set of results, make the experiments reproducible.

To briefly explain the Feature Branch Workflow, the master branch should always represent a fully functional version of the PIC code. Whenever a new feature is to be developed a new feature branch (e.g. Feature 1 in the figure) is created and the user works on that branch until the feature is finished. Then, it is merged back onto the master branch. The user verifies that the master branch executes and is fully functioning before pushing the changes to the central repository (the origin). Ruining the central master branch causes trouble for other users who expect it to be up and running.

Let's consider an example: One user starts implementing a new input settings system for the PIC code (Feature 2). At the same time another user starts revising the field solver (Feature 3). Each user can make as many commits as desirable within their feature branch for the sake of backup. The input system is finished first and its feature branch is merged back onto the master branch before being deleted. It doesn't matter that another user has edited parts of the field solver because that's in another branch. The master branch is now a fully functional PIC code with new input system but with the old solver left intact. Once the new solver is finished, it is merged back to the master. But the master has changed since the revision Feature 3 is built upon. These changes, however, most likely affect other files and Git will be able to seamlessly merge only the appropriate lines changed during Feature 3 development. If uncertainties occur, Git will ask the Feature 3 developer to do some manual work to properly merge Feature 3 with the master branch. Feature 2 will not be overwritten.

Feature branches normally only exist locally. If desirable, they can be pushed to the central repository (the origin) to make them accessible from several computers (for instance for collaboration). The origin should be kept clean, however, meaning that someone must be responsible to delete the branches after merging ensuring that only a few branches exist centrally. Only the repository administrator has the privilege to delete branches and cleaning the origin so other users

[1] See more about various Git workflows here: https://www.atlassian.com/git/tutorials/comparing-workflows .

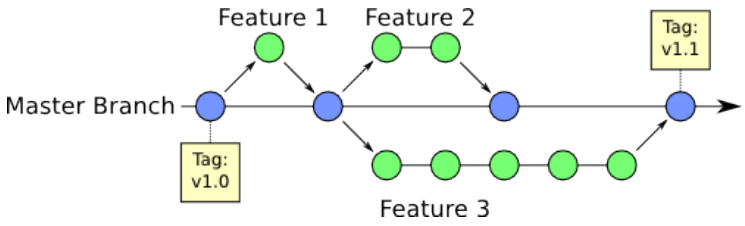

Figure 1: Illustration of Feature Branch Workflow

should ask for permission before pushing new branches to the origin. Moreover, the branches pushed to origin should have globally understandable names.

Finally, some revisions can be tagged with a version number, making it easier in publications to refer to a specific revision of the code. This should only be done with revisions on the master branch.

As a summary: feature branches can have many revisions allowing the developer to go back in case of mistakes, for backup, and for sharing code with other developers. The master branch is holy and only fully functional features should be merged into it.

If this section is unclear or more advice on Git is needed please refer to the "Introduction to Git for `mn-fysrp-pic`" available in `mn-fysrp-pic/docs/git.pdf` before making any changes to the repository.

3 Setting up a Local Copy

To access the repository and be able to make changes you need to set up a local copy. To get access you need an SSH key-pair (public and private keys). Unless you already have that you can run the following command[2]:

```
ssh-keygen -t rsa -b 4096
```

This generates the following files:

- Private key: ~/.ssh/id_rsa
- Public key: ~/.ssh/id_rsa.pub

The private key is private (hence the name) and should under no circumstance be shared with others. It is what you use to access the remote server. The public key cannot be used to log on to a remote server but is rather used by the remote server(s) to verify that an log on attempt is in fact you (i.e. that you have the private key).

Rename a copy of your public key to <username>.pub and mail it to the repository administrator (sigvald.marholm@fys.uio.no) who has to accept your request and forward the message to the UiO IT department.

Next, configure your local git user like this:

```
git config --global user.name '<username>'
git config --global user.email '<email>'
```

[2]http://www.uio.no/tjenester/it/maskin/filer/versjonskontroll/git.html .

Go to the folder where you'd like your local copy (typically your home directory), and clone the central repository (origin) like this:

```
git clone gitolite@git.uio.no:mn-fysrp-pic
```

A new folder with the name mn−fysrp−pic will be created. This is your local working copy.

To get access from other computers you have to copy your private key to ~/.ssh/id_rsa on those computers as well, and run the configuration and cloning steps there as well. You should typically have a local copy on your work station as well as on a supercomputer. Perhaps also on your private laptop if you wish.

4 Folder Structure

For working with the PIC code, your home folder should have the following sub-folders (unimportant details omitted):

- mn−fysrp−pic
  - DiP3D
    - template
    - src
    - lib
  - docs
- local_data
  - template
  - YYMMDD_<simulation description 1>
  - YYMMDD_<simulation description 2>
  - ...

mn−fysrp−pic is the local working copy of the Git repository and is obtained by cloning the central repository as described in the previous section. mn−fysrp−pic/docs contain the documentation of the repository (this document) as well as auxiliary files used to create the documentation.

The source code for DiP3D is located at mn−fysrp−pic/DiP3D/src. The primary task of the repository is to act as a Version Control System (VCS) for the code (.c-files) within this folder. The repository should not track object (.o) files, compiled and linked executables, binaries or similar. VCSs like Git only needs to keep track of the lines changed in text files which makes them very efficient. Other files such as executables and object files carry no real information to the programmer and must be re-stored in entirety every time it changes (after each compilation). Many such files can make the repository heavy. It also clutters the repository with unnecessary changes each time someone recompiles the whole program, causing unnecessary Git conflicts.

Simulation data files should also not be part of the repository. They are also incredibly large and there is also no reason to have version control on them; once a simulation is successfully run, and maybe even used in publications,

it should be considered static. Simulations and their input files are also not, strictly speaking, part of the program.

Third party libraries shipped with the code are stored in mn−fysrp−pic/DiP3D/lib.

The local_data folder is created as follows:

```
cd mn-fysrp-pic
./setup_folders.sh
```

All the simulation-specific files are stored in sub-folders in local_data in order to separate it from the repository. Each simulation has exactly one sub-folder, which should be named YYMMDD_<simulation description> where YYMMDD is the date in reverse order. The sub-folders will contain the simulation results, as well as all the information necessary in order to make the numerical experiment reproducible: input files, job script, and information on exactly which Git revision of DiP3D was used to generate the result.

After simulation is done, the simulation data sub-folders should be copied to some_server.uio.no/path/to/DiP3D_data/ (TBD: folder not created yet). This server is automatically backuped by the IT department.

The local_data/template folder contains example input files which serves as a starting point for making new simulation results. It is simply a copy of mn−fysrp−pic/DiP3D/template. If for instance the format of the input files change, the template in the repository can be updated and setup_folders.sh can be run again to renew local_data/template. All previously existing files in local_data/template will then be deleted, but all other files in local_data will persist.

5 Running a Simulation

Running a DiP3D simulation while keeping folders clean is easily done as described in this section.

First, you should checkout from Git the revision of DiP3D you'd like to use. Next, a new simulation data folder must be made. Here, we use the template as a starting point (a previously run simulation would also work):

```
cd local_data
cp -r template 150611_test_simulation
cd 150611_test_simulation
```

For the time being, the template contains the input files input.txt and sphere.txt which is edited according to the simulation in quest.

5.1 Abel Supercomputer

Next, to execute the job on Abel the job script start_abel is edited according to the users demands (with respect to resource usage and such) and executed:

```
sbatch start_abel
```

The job script copies all DiP3D source files and input files to Abel's scratch area before building the source and executing the simulation. This is the procedure recommended by USIT since the scratch area is faster. The simulation results are copied back to the data sub-folder after execution. This is true even

if the program is terminated unexpectedly. The job scripts also generates a file called execinfo.txt which contains information about exactly which Git revision of DiP3D was used to produce the results, as fetched from Git itself. The file ompiinfo.txt contains the output of the bash command ompi_info showing information on versions of OpenMPI, compilers and miscellaneous. SLURM also writes its output file to this folder.

Abel will automatically delete the scratch folder including all object files and executables. Neither the repository nor the local_data folder will be cluttered with these files. Finally, you should take a copy of the data sub-folder to some_server.uio.no/path/to/DiP3D_data/ (TBD: folder not created yet)

5.2 Desktop Computer

On a plain desktop computer the code can be run without MPI like this:

```
./start_plain.sh
```

start_plain.sh creates a scratch folder within the data folder which acts the same way as scratch on Abel. This folder is deleted if the execution script is successfully run. The information files are generated also in this case.

5.3 Future Changes

As described above, it is necessary to make a copy of the program for each simulation. The reason for this is that the program looks for input files in a fixed relative path to the executable, and stores the output files in a fixed realtive path to the executable. This is about to change. Taking input file as an argument with possibility to specify output directory is in the making which will make execution of DiP3D much more convenient.

In future revisions, the DiP3D program is to be compiled simply in it's source folder. Object files and executable will be omitted from the repository by using .gitignore, and the program can be executed on input files on a completely different location. This will render the currently described folder structure and execution procedure obsolete. Until then, bear with me.

6 Coding Practices

The following conventions are used:

1. Use tab for indentation, not spaces. One tab should be set equal in size to 4 spaces in editor.

2. Lines should not be wider than 80 columns/characters. Tip: Configure your editor to show an edge after column 80.

3. Variable and function names should be intuitive rather than short. wi1j1k is an example of a bad name. Possible exceptions are:

   - Temporary variables declared within a very short scope. The content should be explained next to the declaration of the variable or otherwise obvious.

- Variables with a mathematical origin that are commonly understood within the context of PIC codes. For instance, Ng is commonly used for number of grid points. x is commonly used for position. Such variables should be used a lot (or within a short scope) to justify having a short name. A rarely used variable should have a longer more descriptive name.

4. Variable and function names use underscore as delimeter (set_value()) rather than camelCase (setValue()).

5. Brackets occur on same line as function declaration. Correct:

```
void my_function(){
        if(a){
                ...
        } else {
                ...
        }
}
```

   Wrong:

```
void my_function()
{
        if(a)
        {
                ...
        }
        else
        {
                ...
        }
}
```

6. The code should compile without any errors or warnings. Tweaking compiler options in makefile just to achieve this is not permitted.

7. Globally available functions should be declared in a header file. Local functions (within one .c-file only) should be declared at the top of the .c-file before any functions are defined (mind the difference between function declaration and function definition). Local functions might as well be declared as static for performance.

8. Multi-line comments should be written the following way:

```
/*
 * COMMENT
 * COMMENT
 */
```

   If desirable, the document can be visually divided into sections by comments formatted this way:

```
/*****************************************
 * NEW SECTION
 *****************************************/
```

The developer may have an indentation before the section comment to make it align properly within routines, but the last characters on the horizontal *-lines should be on column 80.

9. Doxygen is used to auto-generate documentation of the code. Doxygen comments are written the following way (note the extra * on the first line):

```
/**
 * Doxygen interpretable comment
 */
```

Each file starts with a Doxygen-comment looking something like this:

```
/**
 * @file                input.c
 * @author              Sigvald Marholm <sigvaldm@fys.uio.no>
 * @copyright           University of Oslo, Norway
 * @brief               PINC main routine.
 * @date                11.10.15
 *
 * Functions for parsing input to PINC.
 * Replaces old DiP3D input.c file by Wojciech Jacek Miloch.
 */
```

All functions are documented by a Doxygen-comment looking something like this just in front of the function declaration. Example:

```
/**
 * @brief       Parse PINC's input argument and input file
 * @param       argc            Argument count
 * @param       argv            Argument vector
 * @return      void
 *
 * This function performs a sanity check of argc and argv and reads the
 * specified input file. It performs the necessary sanity checks of its
 * data and stores the values. It also computes derived values.
 * In case of failure it prints an error to stderr and terminates PINC.
 */
void parse_input(int argc, char *argv[]);
```

For functions declared in header-files, documentations goes in header files as well. For local functions, documentation goes in .c-file along with function declaration at the top of the document. The @-tags used in these examples are self-explanatory and are used by Doxygen to auto-generate information about all files and functions.