Introduction to `mn-fysrp-pic`

Sigvald Marholm

14.10.15

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 .

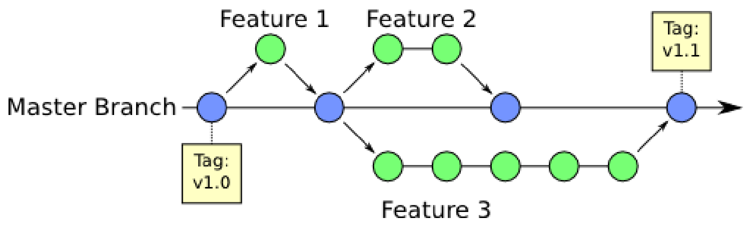

Figure 1: Illustration of Feature Branch Workflow

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 and more advice on Git is needed please refer to `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 authorize when logging into the remote server. The public key cannot be used to log in but is used by the remote server(s) to verify that you have the private key.

Rename a copy of your public key to `<username>.pub` where `<username>` is your UiO username, and mail it to the repository administrator (sigvaldm@fys.uio.no) and wait for approval.

Next, configure your local git user using these commands:

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

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

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

```
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.

If you want access from another computer (e.g. supercomputer) you have to copy your private key to `~/.ssh/id_rsa` on that computer, and run the configuration and cloning steps there as well.

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`
    - `doc`
  - `docs`
  - `proof of concept`
- `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 (including this document) along with 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. For this reason these file types are added to `.gitignore`.

Third party, non-standard, libraries are shipped with the code in order to ensure consistent compilations amongst developers and on supercomputers. They are stored in `mn-fysrp-pic/DiP3D/lib` the way they are provided by the vendor, including necessary legal information.

Compilation of DiP3D is performed by running the makefile in `mn-fysrp-pic/DiP3D`. This also executes the makefiles of third party libraries, and auto-generates documentation of all C-functions in `mn-fysr-pic/DiP3D/doc` using Doxygen. Both HTML and LaTeX-documentation is generated.

The `mn-fysrp-pic/proof of concept` folder is used to test smaller parts that shall later be part of the program.

Note: The folder structure is partly an inheritance from old DiP3D which looks for files with a hard-coded path two steps up. A new main routine is being developed and once that is finished the folder structure will be flattened and simplified. The program will also likely be named PINC (Particle-IN-Cell).

Simulation data files should also not be part of the repository. They are 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. For this reason the `local_data` folder can be 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.

Note: The local_ data folder will soon be deprecated when the new main routine is finished.

5 Running a Simulation (Deprecated)

Note: Ignore this section. It will be replaced once PINC is ready.

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.

6 Coding Practices

The following conventions are used (not complete):

6.1 Program Structure

- No global variables allowed.
- Globally available functions of course must be declared in .h-file. Functions which are only used locally wihtin one .c-file should be declared in the top of that file (before any function definitions). Mind the difference between declaration and definition. Local functions should also be defined `static`, but beware not to over-use `inline`.
- Do not pass big objects to functions, but rather pointers/addresses.
- Brackets do not get separate lines. Correct:

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

Wrong:

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

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

- Whenever a function takes in a pointer to a variable that is not to be modified (e.g. a string) it should be declared const:

```
void function(const char *str);
```

  This serves two purposes: (1) it prevents accidentally changing the content of the variable inside the function. (2) It tells other developers that this function will not change the content of this variable. If a local copy is to be made inside the function it needs to be recast to allow editing:

```
char *temp = (char*) str;
```

- DiP3D (and many others) output way too much info to terminal. Computed values generally should not be output to terminal but to file. Only status messages, warnings and errors belong to the terminal. Use the PINC-specific command `msg()` to do this rather than `printf()` in order to ensure a consistent output behaviour. See documentation of `msg()` in Doxygen.

6.2 Naming

- I have not yet determined whether to use `underscore_names` or `camelCaseNames` . I always used underscore before, but have become fan of camelCase the last couple of years. Most common in C, however, is underscore. We'll see.

- Variable and function names should be intuitive rather than short (e.g. `kinetic_energy` rather then `knerg`). However, sometimes it is reasonable to use short names. Examples:

  - `temp` is a perfectly understandable name for a temporary variable inside a sufficiently short scope.

  - `i` is a perfect name for a loop counter.
  - Established C conventions such as `argc` and `argv`.
  - Variables with a mathematical origin that are commonly understood within the context of PIC codes. See below. Such variables should be used a lot (or within a short scope) to justify it having a short name.

Indices are used for many things in numerical simulations, especially so in particle-mesh codes. This is an attempt to get consistent use of indices both within the code and within mathematical expressions:

- Time step is denoted n (`n`). The number of time steps is N_t (`Nt`).
- Particles are counted using i (`i`). The number of particles is N_p (`Np`).
- Particle species are counted using s (`s`). The number of species is N_s (`Ns`). The number of particles within each specie is denoted N_{ps} (`Nps`) such that $N_p = \sum_s N_{ps}$.
- Particle position is given in small case, e.g x_i (`x[i]`) (considering 1D for this example. For 3D (x_i, y_i, z_i) will likely be stored in one variable).
- Dimension is denoted d (`d`) and N_d (`Nd`) is the number of dimensions.
- Nodes in the grid are labelled (j, k, l) or `j`, `k` and `l` in C. i is not used to avoid confusion with particle count. The number of grid points are denoted is N_{gx}, N_{gy}, N_{gz} and $N_g = N_{gx}N_{gy}N_{gz}$, or alternatively, N_{gd} such that $N_g = \prod_d N_{gd}$ and correspondingly in C.
- The position of grid nodes is denoted by capital letters to distinguish them from particle positions (e.g. X_j or `X[j]`).
- Grid quantities are indexed using a single index p such that $p = j(N_{gy}N_{gz}) + kN_{gz} + l$.

Inside scopes not dealing explicitly with particles it is of course completely valid to use i, j to count other stuff.

6.3 Comments and Documentation

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

```
/*
 * This is my comment.
 * This is also my comment.
 */
```

- If desirable, the document can be organized using sections and titles. Sections and titles look like this:

```c
/**************************************************
 * THIS IS A SECTION
 **************************************************/

void main(int argc, char *argv[]){

        /*
         * THIS IS A TITLE
         */

        int a = function();
```

  The last character on the section comment should be on column 80.

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

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

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

```c
/**
 * @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:

```c
/**
 * @brief Brief one-line description of function
 * @param       in      Integer in.
 * @param[out]  out     Result is stored in this variable.
 * @return      void
 *
 * More extensive description.
 */
void function(int in, int *out);
```

- As a rule of thumb: Comment what functions does, not how. If “how” needs explanation probably the function should be rewritten. “What” is sufficiently documented in the Doxygen comment.

6.4 Formatting and language

- English is the working language.
- Use tab for indentation, not spaces. One tab should be set equal to 4 spaces.

- Output printed to terminal should be no more than 80 columns/characters as this is the default terminal width.
- Lines in source files should be no longer than TBD.
- Whenever a date is to be written, for instance in a Doxygen comment, it is written dd.mm.yy.
- Try to avoid incomplete sentences and poor language in comments, especially in Doxygen comments. End sentences with period.