Introduction to `mn-fysrp-pic`

Sigvald Marholm

26.10.15

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. **??**. 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 .

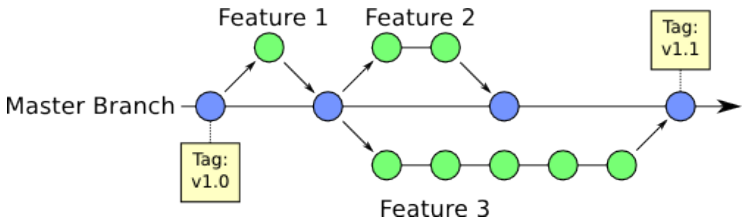

Figure 1: Illustration of Feature Branch Workflow

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

The `mn-fysrp-pic` repository has the following folder structure (unimportant details omitted):

- `mn-fysrp-pic/`
  - `doc/`
    - `doxygen/`
    - `html/` (gitignored)
    - `introduction/`
    - `latex/` (gitignored)
    - `introduction.pdf`
  - `lib/`
  - `old/`
  - `proof of concept/`
  - `src/`
    - `main.c`
    - `pinc.h`
  - `check.sh`
  - `input.ini`
  - `makefile`
  - `pinc` (gitignored)

`doc` contains documentation of the code and the repository. That includes this document (`introduction.pdf`) along with a similarly named folder containing the LaTeXsource files used to create it. `html` and `latex` contains documentation of the code in HTML and LaTeX, respectively. This documentation is not, strictly speaking, part of the repository, but rather is auto-generated by Doxygen from the files in the repository when building the code. The `doxygen`-folder includes auxiliary files needed by Doxygen to do this.

Third party libraries are shipped with the code in order to ensure consistent compilations amongst developers and on supercomputers, and to avoid dependency issues. They are put under the folder `lib` uncompressed but otherwise as provided by the manufacturer, i.e. uncompiled and with legal information files/licenses.

The source code is located at `src` and it is the primary task of the repository is to act as a Version Control System (VCS) for the code (.c-files and .h-files)

within this folder. The repository should not track object (.o) files, compiled and linked executables, binaries or similar (also true for third party libraries). 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` which means that they will not be committed to the repository.

`main.c` is the main-routine, whereas `pinc.h` is the main header file declaring a framework for all PINC modules. For instance the datatypes for storing grid quantities and particles are declared in `pinc.h` such that different modules can be developed that manipulates them consistently. When a new module is developed, for instance a multigrid solver, it is implemented in two files, `multigrid.h` and `multigrid.c`. The framework files (e.g. `pinc.h`) should not depend on the modules in case they are to be replaced in the future. `pinc.h` also makes accessible functions for input and output handling.

To build the code call `make` from `mn-fysrp-pic`. This will build the executable `pinc` along with updated HTML and LaTeX documentation. When adding header or source files to the code, their name must be appended to the `HEAD_` or `SRC_` variable in `makefile`, respectively. (Beware that many files in `src` are inheritance from the old DiP3D code and are not actually in use. Which ones are in use is evident from `makefile`). When adding third party libraries to the project the `makefile` must be updated to compile and link the libraries such that other users do not have to worry about them. `check.sh` is just an auxiliary file used by the makefile. The program is built as `pinc` and `input.ini` is a template input file for PINC.

The folder `proof of concept` may be used to test smaller pieces of code, for instance to check the fastest way to implement something. If, for instance, it was found that one way of implementing something is faster than the other, it might be desirable to keep this experiment for future evidence. This folder is *not* intended to store lots of old rubbish, it should exclusively be something that is clean enough to re-open and and (with little effort) refer to. The folder `old` may *temporarily* hold old files which are to be deleted once a replacement is developed.

Finally, simulation data files should 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.

5 Coding Practices

5.1 Naming Conventions

We use the camelCase-convention rather than underscore. E.g:

```
1 int myVariable = 0;
2 void myFunction();
```

Names should be intuitive rather than short. On the other hand, too long names are bothersome too. Obvious abbreviations are okay. Common abbreviations and names are okay too even if hard to read simply because their commonness makes them more likely to be understood than easier but unfamiliar names. Examples:

- `knErg` is a worse name than `kineticEnergy` which is a worse name than `energy` (if kinetic energy is the only kind of energy that makes sense).
- `wij1k1` is a horrible name.
- `temp` is a perfectly understandable name for a temporary variable inside a sufficiently short scope. No need to call it `kineticEnergyFromPreviousTimeStep`. `temp2` however is usually a sign that you need to clean up.
- `i` and `j` are perfect names for a loop counters. No need to call it `loopCounter`. Make sure to avoid confusion with common uses of `i` and `j` in PINC, though.
- `argc` and `argv` are okay since they are understood by any C programmer.
- `pos` is okay since it's obviously a contraction of "position".

Generally speaking, the larger the scope a variable or function has the more important is its proper naming.

5.1.1 Grammar

Moreover we have the following grammar conventions:

- `nElements`. "n" in the beginning of a variable name like this is read like "the number of". The word(s) that come after "n" should be plural. You don't say "the number of element".
- In many cases it is a good idea that arrays have plural names. Consider this example:

```
int numbers[100];
for(int i=0;i<100;i++){
        int number = numbers[i];
        // Do something with "number"
}
```

- Because of the above point, datatypes (and structs) should have singular names. Consider these examples

```
// CORRECT (if the struct contains only one particle)
Particle particles[100];

// WRONG: This is just confusing.
Particles particless[100];

// CORRECT. Each element is a population of particles.
Population pops[100];
```

Table 1: Suggested names in PINC

	C	Math
time step number of time steps	`n` `nTimeSteps`	n N_t
dimension number of dimensions	`d` `nDims`	d N_d
specie number of species	`s` `nSpecies`	s N_s
particle index number of particles	`i`	i N_p

node index within local MPI subdomain
MPI node index
node index in global grid
	`p=func(j,k,l)`
	`P=func(J,K,L)`
	`gp=func(gj,gk,gl)`
		$p = f(j,k,l)$
		$P = f(J,K,L)$
		$p' = f(j',k',l')$

5.1.2 Structs

Structs are named with a capital first letter to indicate that it is a struct:

```
1 typedef struct{
2         ...
3 } MyStruct;
```

More precisely, we actually do *not* give the struct a name, but rather name its datatype using `typedef`. Naming the struct would look something like this:

```
1 // WRONG
2 typedef struct MyStruct{
3         ...
4 } MyStruct;
```

or:

```
1 // WRONG
2 struct MyStruct{
3         ...
4 };
5 typedef struct MyStruct MyStruct;
```

Naming *both* the struct and the datatype means that the same variable could be declared using two notations and we strive for consistency:

```
1 MyStruct newInstance;          // CORRECT
2 struct MyStruct newInstance;   // WRONG
```

The only exception I can think of when naming the struct itself is necessary is when using opaque datatypes, if that will ever be relevant.

5.1.3 Indices

In numerical codes there are often many indices. As an effort to avoid confusion one should not use one variable name for loop counter of particles one place and for grid another place. Tab. **??** shows names which are to be used for loops through grids or particles along with comparisons to how they could be written mathematically.

Consider for instance iterating through all species:

```
1 |         for(int s=0;s<nSpecies;s++){
2 |                 // do something
3 |         }
```

The grid quantities are stored in a flat manner in one-dimensional arrays. Thus a conversion between three-dimensional indices (e.g. (j, k, l)) and a flat index (e.g. p) is required. The table lists both flat and three-dimensional indices. If nested loops need to iterate across the same index simply use a double letter for the inner loop (e.g. `ii` for particles or `gjj` for global grid).

In functions not dealing with particle or grid quantities it is perfectly valid to use e.g. `i` or `j` as loop counters for generic purposes. Please also see the documentation of the `Grid` and `Population` structs to get a better understanding of naming conventions.

5.2 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 translation unit (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`. `inline` can be used the way it is intended but should not be exaggerated and only works on local functions!
- Do not copy large chunks of data unnecessarily, i.e. by passing structs to functions. The preferred way is to initialize data at one place and rather pass along the address (either through call-by-reference or through pointers).
- Brackets do not get separate lines. Correct:

```
1 | void function(){
2 |         if(a){
3 |                 ...
4 |         } else {
5 |                 ...
6 |         }
7 | }
```

Wrong:

```
1  | void function()
2  | {
3  |         if(a)
4  |         {
5  |                 ...
6  |         }
7  |         else
8  |         {
9  |                 ...
10 |         }
11 | }
```

- 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:

```
1 | 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:

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

- Many numerical codes output way too much information to terminal. Computed auxiliary values generally should not be output to terminal but to file. Only messages on progression status, 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. Auxiliary variables and more extensive messages can be printed to file using `fMsg()`. See documentation of `msg()` and `fMsg()`.

- Do not make new datatypes hiding pointers. Counter-example:

```
1 | typedef double* doubleArr;      // WRONG
```

  This hides what is actually going on and therefore makes it more difficult for other programmers to understand. C-programmers need to be confident about their pointers. The only permitted exception is opaque datatypes (if that is ever needed).

5.3 Comments and Documentation

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

```
1 | /*
2 |  * This is my comment.
3 |  * This is also my comment.
4 |  */
```

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

```
/***********************************************
 * 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):

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

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

```
/**
 * @file                main.c
 * @author              Sigvald Marholm <sigvaldm@fys.uio.no>
 * @copyright           University of Oslo, Norway
 * @brief               PINC main routine.
 * @date                11.10.15
 *
 * More extensive description
 */
```

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

```
/**
 * @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 exhaustive explanation probably the function should be rewritten. "What" should be documented in the Doxygen comment.

5.4 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 132 columns/characters, although doxygen comments should stop at column 80.
- 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.