Introduction to `mn-fysrp-pic`

Sigvald Marholm

25.02.16

1 Introduction

The `mn-fysrp-pic` Git repository holds the Particle-In-Cell (PIC) code PINC (Particle-IN-Cell) belonging to the 4Dspace project and the Plasma and Space Physics group at the Physics Department of UiO. In order to keep the code and its 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 before contributing with any code, or before committing to the repository. Repeated failure to do so may result in reduced privileges or exclusion from the repository. To get access it is sufficient to read Sec. 2, but it is expected that the developer is familiar with the rest before starting to develop.

2 Getting Access

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[1]:

```
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:

```
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. By default it shows the files in the master branch. Typically, the most recent work is in the develop branch. See Sec. 3.

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

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

3 Workflow

The `mn-fysrp-pic` repository utilizes the Git version control system (VCS) and a workflow, or model of how to work with the Git repository, called Gitflow. There are many benefits of Git and Gitflow. When using Git (or VCS's in general), many developers can work on the code simultaneously and thus benefit from each others work almost immediately, while keeping the change history of the code. Gitflow adds to this a guarantee that developers won't inhibit each others work by creating malfunctioning code thereby preventing other users to run the code. With Gitflow, each user "isolates" a copy of the code and work on that part until it is fully functional before merging it back into the main code without overwriting the work of co-developers.

It is near impossible in a modern and active developing environment with multiple programmers to have a well maintained code without a VCS and a decent workflow that everyone follows. If you fail to follow the workflow, someone else will have to struggle with, and clean up your mess. Or more likely, the repository administrator will reject your code and revert the repository to the last sane revision and tell you to fix your errors before merging.

In Git, there are two main branches that exists indefinitely; master and develop. See Fig. 1. The develop branch is where you and your co-developers adds new features. This will hold all the latest functionality. The master branch is reserved for versions that we considered a "finished product" or versions we may refer to in publications. It is rarely updated, and you shouldn't change it unless you've agreed to with the repository administrator. It is holy. Therefore, you probably don't need to worry about the hotfix and release branches depicted at this point either.

3.1 Starting a New Feature

In your everyday work with PINC, what you'll be using is the develop branch, and a bunch of feature branches. Thus when you start working with the repository, you may want to switch (checkout) directly to the develop branch, as that's where the code we're currently working together on is. The first time you do this, the develop branch doesn't even exist locally, and you need to get it from the central server (called the origin):

```
git checkout -b develop origin/develop
```

The next time, you can just switch as follows:

```
git checkout develop
```

You may check which branch you're currently in by

```
git branch
```

When you're going to implement a new feature, you make a branch off of the develop branch. You may first want to make sure you have the most recent version of the develop branch by pulling it from origin. While you are in the develop branch, run:

```
git pull
```

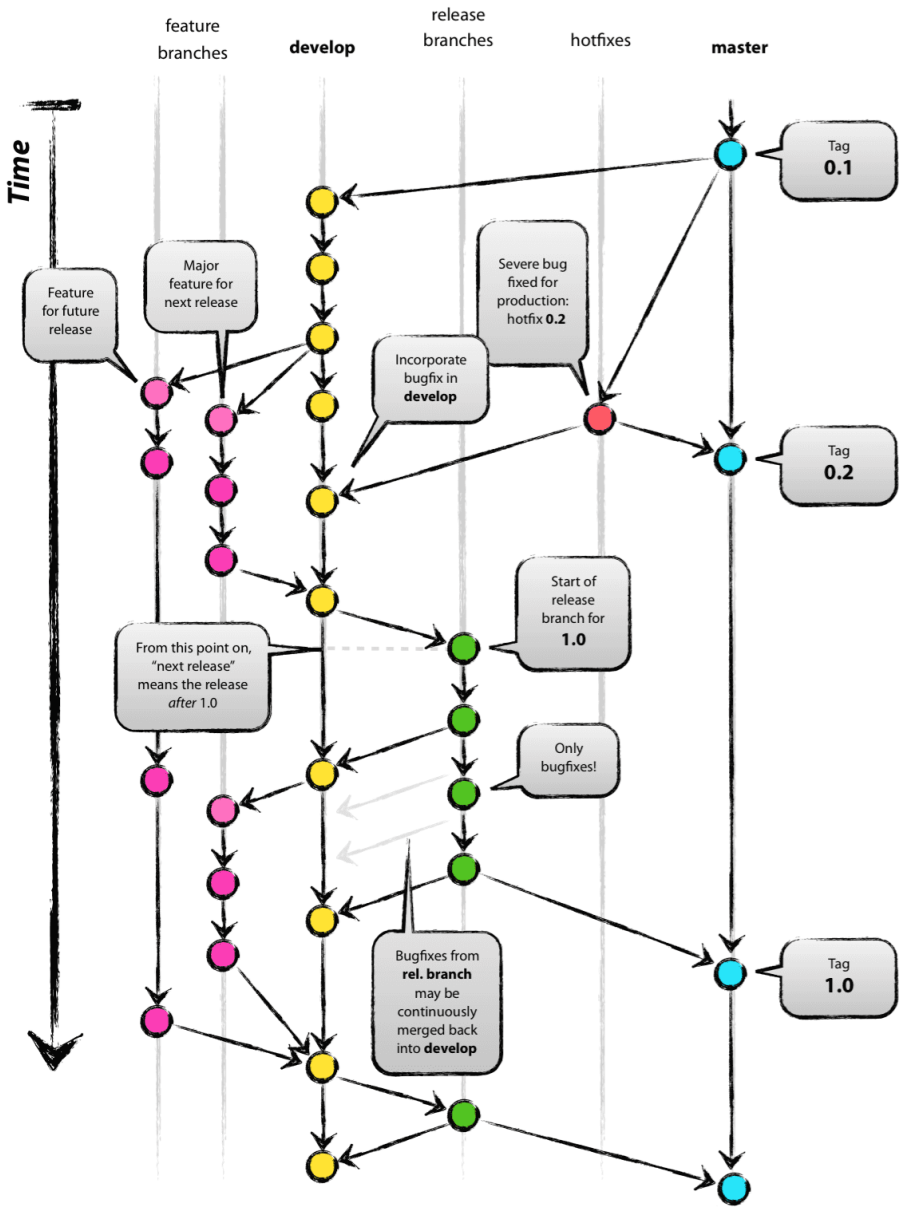

Figure 1: The Gitflow model. Illustration made by Vincent Driessen under Creative Commons BY-SA license, http://nvie.com/posts/a-succesful-git-branching-model.

Then, to create the branch run (while still in the develop branch):

```
git checkout -b myfeature
```

where "myfeature" is the name of your new feature. This should be a short and descriptive name for the feature you are about to implement. Now you implement your feature. A feature branch represents one small task only. You should not develop distinct functionality within the same feature branch. If you're working simultaneously with distinct tasks, you should create distinct branches off of develop, one for each feature. Feature branches should also represent small tasks and should be short-lived, typically a day to a month. The longer a feature branch lives, the more your code will diverge from everyone else's work and you will eventually have to spend a lot of time on manual merging. For short lived feature branches, Git will typically be able to automatically merge your branches.

3.2 Committing Your Work

The changes applied to the code in your working folder is not magically incorporated into the repository. You need to commit your work to the repository to store it. At any later point you can go back to the state of the code at any previous commit. First, you may want to inspect which changes are done. You do this by:

```
git status
```

You will get a list of files that are changed since the last revision (labelled "Changes not staged for commit") and a list of files that are totally new (labelled "Untracked files"). Initially, no files are labelled as "Changes to be committed", meaning that committing at this point will add no changes to the repository. You will have to tell Git which files are to be committed by the following command:

```
git add <list of files>
```

This is called to "stage changes for commit". Running `git status` again will now show the staged files as "to be committed", and you can commit your working folder to the repository by

```
git commit -m '<message>'
```

The argument after `-m` is a message for the commit log. This will be useful if you or someone else wants to go back to a previous state. Note that you may have as many commits as you like. It is a good idea to try to let each commit represent one thing, and write that in the message, rather than just thinking "now I want to make a backup".

If you want to stage and commit all files at once, you can simply run

```
git commit -am '<message>'
```

However, this makes it easy to by mistake commit files that should not go into the repository. For instance files created by your text editor. Therefore, you should *always* run `git status` before committing. Further on, it is a good habit to properly clean up the `git status` before committing, such that only files "to be committed" shows up (unless you actually want to commit only part of your changes). There will often be files that should be deleted. Then delete them. If there are files which you need locally but does not belong to the repository, such as text editor files, add them to a `.gitignore` such that Git ignores them.

Notice that when you commit something in Git, you do not immediately store it to a central server. You only commit it to your local clone of the repository (as represented by .git folders), and you may push it later. More about that later.

You may inspect the commit log simply by the use of `git log`. Several arguments exist for it, for instance, try:

```
git log --oneline --graph --decorate
```

Finally, if you move or remove a file that is previously known by Git (committed), use `git mv` or `git rm` rather than just `mv` or `rm`. Otherwise Git will not understand what's going on and you will have to clean up.

3.3 Finishing Your Feature

When your feature is implemented and tested and compiles *without* errors or warnings (others will use your code), you can merge it back to the develop branch and finish the feature branch by deleting it:

```
git checkout develop        # Switch back to develop
git pull                    # Get most recent develop from origin
git merge --no-ff myfeature # Merge myfeature into develop
git branch -d myfeature     # Finish/delete myfeature
```

On the second line, your feature will be merged into the code base along with new features that others have merged while you were developing your feature. When Git are unable to automatically merge your feature, you will have to resolve merge conflicts by manually going through the code. Git will mark the regions of conflict in the source code for you to find. After resolving conflicts you have to commit again.

Since you're using `-d` rather than `-D` on the last line your branch will not be deleted unless all changes are successfully merged to develop.

Next, all the changes you've done up till now has been committed to your local repository only. To share them with others you need to push the develop branch to the origin:

```
git push           # Do this while in the develop branch
```

And that's basically the whole cycle, which you repeat for every feature.

3.4 Pushing Feature Branches

Using the workflow as described so far, the feature branches (unlike develop) will only exist locally, on your computer. You may, if you want to, push feature branches to the origin. Then you can work on them from several computers, and you will have a backup on the server. The caveat is that there's slightly more details to pay attention to.

If you have a feature branch, say "myfeature", you can push it to the origin by

```
git push -u origin myfeature
```

The `-u` switch is not strictly necessary, but it makes the branch trackable, i.e. it establishes a connection between "myfeature" on your computer and "myfeature" on the origin. Thence, the next time you need to push/pull changes of this branch to/from the origin, you just do:

```
git checkout myfeature # Making this the current branch
git push               # or git pull
```

When your feature is finished, you merge it back into the latest revision of the develop branch just as previously described. However, you should make sure that the latest existing feature branch exist on origin prior to merging and deleting the feature branch. The whole finishing procedure may look something like this:

```
git checkout myfeature
git push                    # Push the latest myfeature to origin
git checkout develop
git pull                    # Get most recent develop from origin
git merge --no-ff myfeature # Merge myfeature into develop
git push                    # Push the new develop branch
git branch -d myfeature     # Finish/delete myfeature
```

The last line will only delete the feature branch locally, and you now need to tell the repository administrator that you want to delete the feature branch on origin (for safety, you don't have privileges to this). Since you pushed the latest feature branch to origin the repository administrator can verify that no data is lost by deleting it.

3.5 Further Reading

You are expected to follow Gitflow. If anything is unclear you can read more about it here:

```
http://nvie.com/posts/a-successful-git-branching-model/
https://www.atlassian.com/git/tutorials/comparing-workflows
```

or contact the repository administrator.

4 Reproducibility

It is currently a long term plan to release PINC as an open source project. Some revisions of PINC can then be tagged in Git with a version number, and authors of scientific publications can specify which version was used, thereby making the results truly reproducible by others, as is the spirit of science. Version tags should only be put on revisions on the master branch.

Moreover, it is a not-yet-implemented feature of PINC that an auxiliary file is output which states which version and revision of PINC was used to generate the results, which versions was used of third party libraries, and which hardware was used. This feature is supposed to fetch the version tag and the revision number automatically from Git.

5 Repository Structure

5.1 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` (this file)
  - `lib/`
  - `proof of concept/`
  - `script/`
  - `src/`
  - `test/`
  - `input.ini`
  - `makefile`
  - `mpinc.sh`
  - `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 LaTeX source 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.

Small non-standard third party libraries are shipped with the code for convenience. They are put under the folder `lib` uncompressed but otherwise as provided by the manufacturer, i.e. uncompiled and with legal information files/licenses. Unfortunately, it has too many drawbacks to include big standard libraries in this folder and perform static linking even though this would've guaranteed consistent executions across computers. See Sec. 6.

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.

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. `old` folders may *temporarily* hold old files which are to be deleted once a replacement is developed.

`test` is the unit testing source files, to be more thoroughly documented at a later point, while `script` is a folder of Python scripts which are/can be used to visualize and interpret data. These are both useful for testing the code and for later starting points when doing simulations. Python scripts may also be used to do parameter sweeps by running PINC with several input parameters.

Finally, simulation .h5-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.

5.2 File Structure

The PINC source files can be thought to belong to one of three categories as illustrated in Tab. 1:

`main.c` is the main-routine, whereas `pinc.h` is the main header file declaring a framework for all PINC modules (pedantically it's not a framework but is thought of as such). It declares datatypes and functions for handling particles and grid quantities along with functions to manipulate them and other building block functions ensuring a coherent code. The definitions are stored in the accompanying .c-files. Modules, for instance a multigrid solver, act upon these standardized data structures, and is implemented in one .h-file and one or more .c-files starting with the name of the module. The framework files should *not* depend on the modules.

Table 1: Source file categorization

	.c-files	.h-files
Main routine	`main.c`	

Framework	`aux.c`
	`grid.c`
	`io.c`
	`population.c`
		`pinc.h`

Multigrid module	`multigrid.c`	`multigrid.h`
Future module	`moduleA.c` `moduleB.c`	`module.h`

6 Compiling and Running PINC

Before being able to compile PINC the following libraries must be installed:

1. OpenMPI
2. GNU Scientific Library (GSL)
3. HDF5 API including parallel support (more on parallel support later)

These are typically installed either through `sudo apt-get install` when available or by downloading the source code and executing something like this:

```
./configure
make
sudo make install
```

The last step is crucial since this makes the system find the library without having to add local modifications to the PINC makefile. While it is permitted to do whatever you like locally, it is not permitted to push files of local relevance only to origin, and it is therefore advised to install the libraries on the system.

Once dependencies are resolved PINC is compiled using the makefile in the repository:

```
cd mn-fysrp-pic
make
```

The makefile installs the libraries in `lib` as well as generating Doxygen documentation. When adding new header or source files these can be added to `HEAD_` or `SRC_`, respectively, in the makefile. Next, PINC is executed as

```
./pinc input.ini
```

where `input.ini` tells PINC which settings to use. More than one process can be run in the normal way using `mpirun` as illustrated below (for two processes):

```
mpirun -np 2 pinc input.ini
```

however, the number of subdomains to use in the domain decomposition must correspond to the number of processes. If, for instance, you have $4 \times 4 \times 4$ subdomains specified in `input.ini`, the number of processes must be 64. This command can be run more simply as:

```
./mpinc.sh input.ini
```

which extracts the number of processes required by PINC and automatically sets the `np` parameter of `mpirun`.

PINC is more kind of a numerical "engine" than a tool for diagnostics, parameter sweeps and so on. To facilitate parameter sweeps performed by external (Python) scripts, parameters in the input file can be overridden through additional command line arguments to PINC, following the following syntax:

```
./mpinc.sh input.ini <section>:<key>=<value> <section>:<key>=<value> ...
```

For instance, the number of subdomains to use is specified as the parameter `nSubdomains` under the section `grid` in `input.ini`. To override the input file and specify $4 \times 4 \times 4$ subdomains, call the following command:

```
./mpinc.sh input.ini grid:nSubdomains=4,4,4
```

Unfortunately, a value for `nSubdomains` must already be set in `input.ini` to be able to override it through the command line (this is due to a limitation of the third party library).

Finally, to erase all build files run

```
make clean
```

7 Tools

This section describes tools that can be useful for developers.

7.1 Editor

Use whichever editor you want (I use Atom) but do not clutter the repository with editor files. Also, make sure the editor settings obey the coding conventions with respect to spaces, tab sizes, etc.

7.2 Local Scribble Area

As the program evolves and `main.c` becomes fully functional, we try to avoid doing small test in that file. However, to have an area to scribble and test on while developing, one can make a local main-file `main.local.c` to mess around with. To compile using this file rather than the official `main.c` file run:

```
make local
```

`main.local.c` is gitignored.

7.3 Doxygen

As already mentioned, the code is documented using Doxygen. Build PINC and the documentation will be at `mn-fysrp-pinc/doc/html/index.html`.

7.4 Unit Testing

To run unit tests, execute the following command:

```
make test
```

which will build a unit test executable `ut` and execute it. How to develop unit tests is briefly described in Doxygen.

7.5 Valgrind

To ensure that dynamically allocated memory is freed use Valgrind. Since OpenMPI generates a lot of false positives these false positives are suppressed by using the PINC wrapper script `valgrind.sh` for running Valgrind instead of just `valgrind`[2]. First, make sure Valgrind is installed on your system. Then, consider the following example:

```
1  void test(){
2          char *ptr = malloc(1048576);
3  }
4
5  int main(int argc, char *argv[]){
6
7          char *str = malloc(16);
8          strcpy(str,"test");
9
10         test();
11
12         printf("%s\n",str);
13         return 0;
14 }
```

Then, compiling and running valgrind as follows:

```
make clean
make COPT=
./valgrind.sh ./pinc input.ini
```

returns (uninteresting details removed):

```
HEAP SUMMARY:
    in use at exit: 1,255,872 bytes in 498 blocks
  total heap usage: 7,251 allocs, 6,753 frees, 14,389,316 bytes allocated

16 bytes in 1 blocks are definitely lost in loss record 88 of 340
   at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x40166D: main (in /home/sigvald/mn-fysrp-pic/pinc)

1,048,576 bytes in 1 blocks are definitely lost in loss record 340 of 340
   at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x40164E: test (in /home/sigvald/mn-fysrp-pic/pinc)
   by 0x401685: main (in /home/sigvald/mn-fysrp-pic/pinc)
```

[2]Since OpenMPI is usually not compiled with `-enable-memchecker` Valgrind will not be able to detect errors in shared memory space. I guess most (all?) mistakes will be eliminated using the PINC Valgrind wrapper, however.

```
LEAK SUMMARY:
   definitely lost: 1,048,592 bytes in 2 blocks
   indirectly lost: 0 bytes in 0 blocks
   possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
   suppressed: 207,280 bytes in 496 blocks

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 53 from 53)
```

This shows that `malloc()` in `main()` allocated 16 bytes which is not freed, and that `malloc()` in `test()` in `main()` allocated 1MiB which is not freed. It also showed that approximately 200KiB which is not freed is suppressed. This is not due to PINC mistake but due to OpenMPI and is therefore not considered an error. Except for the suppressed leaks all leaks should be zero. Feel free to read more about the different leaks online.

The `COPT=` argument in the make-call above is used to turn off compiler optimizations. Without this, the compiler would figure out that the function `test()` does nothing and can be omitted. As a result, only the first error would show up. In fact, the second error does not exist any more after optimizing and therefore pose no problems. Nevertheless, it may be helpful to turn off optimizations when looking for memory leaks.

Finally, beware that running Valgrind degrades execution speed.

7.6 GProf

To get an overview of what is time-consuming in PINC, execute the following commands:

```
make clean
make CADD=-pg
./pinc input.ini
```

`CADD` is used to add additional flags to the compiler. In this case instrumentation code is incorporated into PINC to allow GProf to profile it. Upon execution of PINC the file `gmon.out` (gitignored) is created which can be analysed with GProf. Extensive documentation of GProf is available online but as an example,

```
gprof -bap pinc gmon.out
```

may return

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.04      0.96     0.96        3   320.14   320.14  posUniform
  0.00      0.96     0.00       36     0.00     0.00  intArrMul
  0.00      0.96     0.00       22     0.00     0.00  iniGetStrArr
```

```
  0.00      0.96      0.00       14      0.00      0.00  intArrProd
  0.00      0.96      0.00        7      0.00      0.00  iniGetLongIntArr
  0.00      0.96      0.00        6      0.00      0.00  msg
  0.00      0.96      0.00        3      0.00      0.00  iniGetIntArr
  0.00      0.96      0.00        2      0.00      0.00  iniGetDoubleArr
  0.00      0.96      0.00        1      0.00      0.00  allocGrid
  0.00      0.96      0.00        1      0.00      0.00  allocPopulation
  0.00      0.96      0.00        1      0.00      0.00  tFormat
  0.00      0.96      0.00        1      0.00      0.00  velMaxwell
```

Mind that a significant amount of time is necessary in order for GProf to be able to determine non-zero time.

7.7 Simple Timing using PINC `Timer`

While GProf is suitable to get an idea of the execution of the whole program it has its flaws. One of them is when you want to know rigorously how fast certain operations execute. For this purpose, the PINC `Timer` datatype may come in handy. Consider for instance that you know two different ways of implementing the same thing, one intuitive way using nested loops, and one "clever" way using one loop and a modulo operator. These can be compared as shown below:

```
1   Timer *timer = allocTimer(0);
2
3   int nDims = 3;
4   int L[3] = {1,2,4};
5   double *pos = pop->pos;
6
7   tMsg(timer,NULL);
8
9   for(long int i=0; i<1e7; i++){
10      for(int d=0;d<nDims;d++){
11          pos[i*nDims+d] = L[d]*
                gsl_rng_uniform_pos(rng);
12      }
13  }
14
15  tMsg(timer,"Nested loops method");
16
17  for(long int i=0; i<nDims*1e7; i++){
18      pos[i] = L[i%nDims] * gsl_rng_uniform_pos(rng);
19  }
20
21  tMsg(timer,"Modulus method");
22
23  freeTimer(timer);
```

Then compiling with

```
make clean
make
./pinc input.ini
```

returns

```
TIMER (0): [tot=332.81ms, diff=289.26ms] Nested loops method
TIMER (0): [tot=619.69ms, diff=286.83ms] Modulus method
```

`tot` shows the time since program start-up whereas `diff` shows time since previous invocation of `tMsg()`. `tMsg(timer,NULL)` can be called simply to reset the timer without printing any message.

In this case, we see that both methods perform equally well at approximately 290 ms. The difference is insignificant. It is interesting though, to re-run the experiments with optimizations turned off ("`make COPT=`"):

```
TIMER (0): [tot=397.32ms, diff=359.46ms] Nested loops method
TIMER (0): [tot=956.03ms, diff=558.67ms] Modulus method
```

It is now evident that the intuitive nested loops method by itself is faster but the compiler has optimized both methods such that they perform similarly. This information, however, is purely for educational purpose, as it is no point in optimizing the code with compiler optimizations turned off. The rationale is; if two alternatives perform equally well with compiler optimizations on, choose the prettiest solution. If not, choose the fastest.

Whereas this example was well optimized by the compiler, this may not always be the case. It is therefore highly advised to think through your programming techniques, and when uncertain as to what will perform well, measure it.

As a final note, although the PINC Timer has a very high resolution the accuracy is more influenced by random variations and overhead when measuring short times. A remedy to this is to put the operations to be measured in a loop to get the timer up to the the millisecond range. To get an idea of the resolution, the overhead and the accuracy of `tMsg()` try measuring `tMsg()` itself by running it in a loop[3].

7.8 GNU Debugger (GDB)

GDB may be useful if for instance you want to step through your code, inspecting what's happening to variables as the program run. To be able to use GDB the program must be compiled with the additional `-g` flag as follows:

```
make clean
make CADD=-g
gdb pinc
```

from which point you can use GDB. GDB is well documented online. Beware that sometimes things may look strange due to optimizations, for instance the value of a variable may not exist and show up as "<optimized out>", either because it simply is not needed, or because it is not needed at this point. As usual, you can turn off optimization for debugging purposes by running `make CADD=-g COPT=`.

8 Coding Practices

Herein is described the coding conventions to be used within the project.

[3]Technically the timer is measured at the beginning of `tMsg()` and reset at the end of `tMsg()` such that `diff` actually shows the time between, but not including, calls to `tMsg()`. This gives more accurate readings of what comes in between. On my computer `tMsg()` executes in approximately $5\mu s$ but the overhead of the measurements is only approximately 200 ns. The discrepancy is largely due the fact that printing to the terminal takes time.

8.1 Naming Conventions

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

```
int myVariable = 0;
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.

8.1.1 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 2: 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)`
		$p = f(j,k,l)$
		$P = f(J,K,L)$
		$p' = f(j',k',l')$

8.1.2 Structs

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

```
typedef struct{
        ...
} 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:

```
// WRONG
typedef struct MyStruct{
        ...
} MyStruct;
```

or:

```
// WRONG
struct MyStruct{
        ...
};
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:

```
MyStruct newInstance;           // CORRECT
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.

8.1.3 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. 2 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.

8.2 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.
- 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:

```
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;
```

- 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:

```
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).

- Strive to follow previous coding style to make the code consistent (or suggest improvements).
- Try to make functions work in an atomic way, i.e. that one function does one thing and does it well.

8.3 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:

```
/***********************************************
 * 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. Mind the alignment of the characters. Sections and titles are written in uppercase, normal comments in normal capitalization.

- 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 perhaps the function should be rewritten? "What" should be documented in the Doxygen comment.

8.4 Formatting and language

- American English is the working language.

- Use tab for indentation, not spaces. One tab should be set equal to 4 spaces.
- Try to make output printed to terminal no more than 80 columns/characters as this is the default terminal width.
- Too long lines in source code can obviously be a mess. On the other hand, breaking up long function calls across multiple lines is a mess too. We strive for a compromise. Text block comments should be no more than 80 columns, as should most of the code. Many function calls (e.g. MPI calls) will easily extend beyond 80 columns, and breaking them up only makes the code messier, so don't. The makefile will emit a warning for lines longer than 132 columns.
- 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.