User Guide and overview of the WMCD code

This C code for Wavelet Monte Carlo dynamics (WMCD) was written and developed during the author's PhD. It follows the details of WMCD as set out in

[1] O. T. Dyer and R. C. Ball, *J. Chem. Phys.* **146**, 124111 (2017),

with additional functionality as a smart Monte Carlo algorithm (details to be published) and able to impart affine deformations on the system (e.g. for imposing a shear flow). Other versions of the code with even greater functionality will be made available in due course.

No claim is made about it being an optimised implementation of WMCD, and it is not set out in the tidiest way, having been modified and added to many times. This is especially true of the functions in `WMCD_main.c`, which might benefit from being rewritten. Nevertheless the code works and has been shown to be competitive [1], and this document provides a description of how it works and how to use it.

Brief descriptions of the files provided

`WMCD_main.c`: contains the core functions for the algorithm, about which more will be said later.

`WMCD_input.c`: contains all the functions associated with passing information into the code, e.g. system parameters and initial configurations. These functions will be described in greater detail later.

`WMCD_output.c`: contains all functions that create and write data files.

`WMCD_maths.c`: contains basic mathematical routines.

`WMCD_phys.c`: contains many functions specific to the code including those with physical meaning (e.g. interaction potentials). Unfortunately this ended up a bit of a dumping ground for functions.

`WMCD_transforms.c`: contains all functions associated with coordinate systems and transformations between them. These are used to impart external flows.

`WMCD_header.h`: header file containing declarations of all functions outside of `WMCD_main.c` along with all structures used throughout the code.

`pcg_basic.c` and `pcg_basic.h`: functions and header file for the PCG random number generator. These are unmodified version found in the link given at the start of the files, and are included in line with the license they fall under.

Regarding affine deformations

Affine deformations are achieved by simultaneously maintaining a static lab frame and a frame co-moving with the fluid. The idea is for bead positions (`xyz[]`) to be stored in co-moving coordinates and thus be displaced implicitly by a time-evolving coordinate transform.

This has many repercussions throughout the code, most frequently the need to transform between reference frames (an optimised version of the code with no interest in external flows would benefit from removing all of these). To help distinguish between quantities in the two reference frames, some variable names are appended with `Cart` (intended to indicate a simple orthogonal Cartesian coordinate system) to denote variables in the static frame.

The `main` function

The `main` function's primary role in this code is to declare a bunch of variables, output information and loop over moves. All the heavy work is placed in the `evolve` function.

The key points in `main` are (in line order):

- The seed for the PCG random number generator is set using all of the current time, job number and array number (the last 2 being `argv` inputs, and usually both set to 1 when not running on a cluster).
    - As WMCD relies on random numbers it is important to avoid multiple runs having the same seed, hence why setting the seed uses so many inputs.
    - When debugging, fixing the seed by hand is often useful. The system parameters output files write the seed in case they are needed for rare bugs, but normally setting `int seed[2]={1,1};` does the trick.
- The next ~300 lines declare variables. *Do not touch these unless you know what you are doing!* Many of these are codified versions of analytic results found in Ref. [1].
- The next block constructs the initial system using parameters set in the `input` function. It defaults to placing particles stochastically (with some protection against particle overlap), but can also be set to read configurations in from files with the `eq_config_in` or `cont_config_in` options.
    - These options can be used to choose an initial configuration by hand if desired, but they are set up primarily to use outputs from previous runs.
- Next there is a loop over moves, which includes checks for which data to output and when to do so.
- After that the code outputs some final information (e.g. how long it took to run) and ends.

The `evolve` function

This function contains the meat of the code and it is here that the algorithm itself is coded. Most modifications to the code will change or add to parts of this function.

Full disclaimer: the key parts of the code would become significantly cleaner and easier to navigate by splitting `evolve` into two functions to handle wavelet and Fourier moves separately (suggested names ‘evolvew’ and ‘evolveF’). This would move an if statement into `main` and remove many in `evolve`, as well as cutting out many variables in each new function. If one does embark on this, just be careful that the path through each function matches that through the current function with a given move type.

The function is structured as:

- It starts by declaring variables needed for the move.
    - You might notice there are a lot of arguments passed into `evolve`. I tried to keep this down by using structures and declaring what I can inside.
- Choose between a wavelet or Fourier move.
- Get some of the wavelet/Fourier (w/F) parameters and identify all moving particles.
- Calculate the energies and forces before making the move.
- Get the orientation and amplitude of the w/F move, biased by the forces present.
    - This is where smart MC comes in. Setting `smart_MC = false` will remove the bias.
    - Note also, this version of the code is geared towards computing the deterministic part of the motion, while keeping the stochastic part for actual move updates. That is the purpose of the blocks entitled “Temporary section. Used for deterministic <v.v>”.
        - These blocks are removed in other versions of the code, but if they are wanted again just copy them over along with any required infrastructure (e.g. passing `dr_acc` to `evolve`).
- Find positions of moved particles as per the w/F move.
- Calculate the energies and forces after making the move.
- Use the forces and the change in energy in the MC acceptance test.
- If the move is rejected, return particles to their previous positions.
- If accepted, update all relevant variables.

The `UF_move` function

Despite its length there is not much to say about this function, though it is helpful to understand how it works. Its task is to sum all the interaction energies which can change over the move (i.e. those associated with moving particles), and to calculate the corresponding forces at the same time.

The procedure uses ‘cell lists’, with the system split up into sub-boxes. Each moving particle then looks in its own box and surrounding boxes for any other particles and calculates forces and energies with any particles it finds there.

It does so in a way that avoids double counting, which mostly means only looking at ‘forwards’ boxes (defined by the `casesall` variable), and looking ‘backwards’ wherever interactions in that direction would be missed.

If the particle is part of a chain, the attractive forces holding them together do not need to use this box-search algorithm because you always know those interactions are present. For these the code simply looks at neighbouring particles along the chain (numbered contiguously).

User Guide

As written, the code has only 2 files that need modifying for a given use: `WMCD_input.c` (and in particular the `input` function), and the makefile.

The `input` function

This function serves as the primary interface for the code. From here the details of simulations are set. The details of each option are described in comments in the code and won't be covered here, but here is a quick overview of what is set, grouped by the structs they appear in.

The `System_parameters` struct contains nearly all of the parameters describing the system, including:

- The physical size of the simulation box `L`. Note this only makes a difference to the physics if `periodic==true`.
- The number of particles in the system, split into `Npoly` linear chains of `Nmono` particles connected by bonds.
- `lambda_min` and `lambda_max` set the range of wavelet radii used, while `use_Fourier` can be set to switch Fourier moves on or off. Mostly they are turned off for debugging when you want to focus on wavelets. There is little added cost to using them so this should be set to `true` by default.
- The specific MC test is set by `smart_MC` (whether to bias moves by forces or not. It is usually optimal to set this to `true`), `use_Glauber` (whether to use the Glauber or Metropolis test. If `smart_MC==true` this should be set to `false`) and `Langevin` (this turns off the acceptance test altogether, but only if `smart_MC==true`. In this case the code is a direct solver of the Langevin equation, but it is rare to want to do this as acceptance rates are usually >99% with smart MC).
- `recycle` is a legacy feature whereby wavelet radii and Fourier wavenumbers would be reused if moves are rejected in order to better approximate the desired distribution. With smart MC this is unnecessary and `recycle` should be set to `false`.
- The other variables are named the same as in Ref. [1], and don't need explaining here.

The `Potentials` struct contains all the parameters specific to interaction potentials. Note:

- Potentials are split into bond potentials between connected particles and potentials that act between all particle pairs.
- Each is parameterised by length and energy scales (i.e. only 2 parameters each), with the capacity for only 1 choice of potential of each type.
- The choice of potential in each case is set by a pointer to a function. All possible functions - found in `WMCD_phys.c` but others can be added if desired – will take in the exact same

arguments for this reason: whichever function the pointer points to, it will always pass in the same arguments when called.

Finally there is the `In_out_options` struct which contains boolean variables that simply set which data to output during a simulation, and whether to read in configurations at the start.

All structs used are defined in the `WMCD_header.h` file, in case you want to add anything to these.

The makefile

The code should compile with the GCC or Intel compilers (others probably work but were never used). With the GCC compiler the bash shell command

```
gcc -lm WMCD_main.c WMCD_maths.c WMCD_phys.c WMCD_transforms.c WMCD_input.c WMCD_output.c pcg_basic.c
```

will compile with all the needed files. Any further options, such as output names and optimisations are up to the user.

Since the original aim of WMCD is to be efficient, the Intel compiler might be preferable. For this use

```
icc -O3 WMCD_main.c WMCD_maths.c WMCD_phys.c WMCD_transforms.c WMCD_input.c WMCD_output.c pcg_basic.c
```

again with options as desired (here the optimisation option `-O3` is present).

Both the gcc and icc commands are included in the makefile provided.

When actually running the executable, called `WMCD.out` for example, you will need to pass in several inputs. Schematically, it would look like:

```
./WMCD.out {Job ID} {Batch ID} {Continue ID} {Option}
```

Job ID: involved in setting the random number seed and enters output file names. This is usually the job number assigned on a cluster, but if running a single job any value is fine (see Continue ID).

Batch ID: similar roles to Job ID, again typically set to 1 on single runs but it is not important to do so in this case. In a batch of jobs sent to a cluster, this should be set to the job number *within* that batch to distinguish the output files of each job.

Continue ID: if set to 0, Job ID will be replaced by the seed in output file names. This is useful when there are no Job IDs to use. When this does not equal 0, whatever you set for {Job ID} will be used in the file name, which is useful if you want to continue gathering data from a previous run and append to the same files, but dangerous otherwise.

Option: passed into the `input` function and is generally used for any variable you want to vary across a single batch, which probably runs off the same executable. What you do with it is up to you.

Additional options can be added if desired.

Modifying/adding to the code

This section highlights the procedure for a few simple and likely changes one might want to make to the code.

Different interaction potentials

Adding new interaction potentials is fairly straightforward because all functions are written using a pointer to a potential function rather than a specific function. Consequently the full list of required changes is simply:

- Write the function calculating the potential energy and 'force' (strictly, if the force is written in terms of the vector between the interacting particles as …., then what should be calculated in the function is ….).
    - Make sure the arguments are the same as for functions for other potentials.
    - For consistency place this in `WMCD_phys.c` with the other potentials, and remember to declare the function in `WMCD_header.h`.
- In the `input` function point the potential pointer to the new function.
    - An example in the current code is `params_input->EV_fn = &EV_WCA;`
    - Ensure you use the correct pointer for the type of potential. `EV_fn` is for general inter-particle interactions (typically excluded volume, hence 'EV') and `bond_fn` is for potentials holding beads together.
- Set potential parameters as desired, but keep in mind the range of the potential if it is for `EV_fn`. For the energy calculation to be valid the maximum range should not exceed the value of `sub_box_size` (set at the top of the `input` function).

Choosing file names and directories

The function `get_directory_and_suffix,` found at the top of `WMCD_output.c`, sets both the directory in which to read/write data and the string of system parameters appended to file names to identify data with (the 'suffix'). Both of these can be chosen to fit one's needs, but be aware that the `batch` and `run_ID` variables should always be included in the suffix to avoid different simulations writing to the same files.

Beyond the `get_directory_and_suffix` function, each output function specifies a sub-directory and the start of the file name for that given output. These should be modified as required.

Adding new outputs

Exactly how to add new outputs will depend on what you want from them and when you want them outputted. It may nonetheless be helpful to explain the approach taken for the existing outputs.

1. The function is written in `WMCD_output.c` and declared in `WMCD_header.h`.
2. A boolean variable is added to the `In_out_options` struct and a line is added in the `input` function setting this to true or false.
3. A corresponding string is added to the 'OUTPUTS' printf call at the end of the `main` function. (This is not strictly necessary but it helps keep track of what's being done.)
4. The new output function is called in the loop over moves in the `main` function, conditional on both whether the associated boolean is true and whether the chosen time has passed since the last output.
    1. As written there are multiple different output times in use (a number of moves is equivalent to a length of time).