Here we describe some standard approaches to developing molecular simulations that will help your codes be reusable, clear, and fast. Why should you care to implement these?

- They will facilitate your code development so that you don't make bad decisions early on that later require significant re-coding.
- The will reduce the possibility of bugs.
- They will make it easy for you to revisit and modify your code after some time has passed, in case you need to re-evaluate a result or want to extend your earlier work.
- They will enable others interested in your work to use your code. If you publish novel results based on your simulations, there is a very real possibility that other computational scientists might want a copy of your code.

Make your bottleneck routines *fast*, everything else *clear*

There is absolutely no need to optimize every single part of your code. In fact, optimization will often come at the expense of code legibility and reusability, and can be a big source of bugs. A good strategy is to make your code as simple, straightforward, and clear as possible, optimizing only those steps that are the most expensive.

Here is a good rule of thumb: if you can't get at least twice the speed out of a particular optimization, it's not really worth implementing it if it sacrifices clarity or generality.

Techniques for making things fast

Code only the most expensive routines in Fortran, and keep everything else in Python. This usually means the pairwise interaction loop, and perhaps also any integration steps.

Profile your Python code to identify where the bottlenecks are, and then optimize these routines or rewrite them in Fortran. Python's profile module will automatically provide a breakdown of the CPU time required by each algorithm.

In Fortran, keep in mind the relative expense of different mathematical operations. Computations using integers are generally very fast. Floating-point calculations are the biggest bottleneck. Addition, subtraction, and multiplication are fast operations. Division and exponentiation are slightly slower. The most expensive routines correspond to sqrt, exp, log, ln, sin, cos, tan, and related functions. If expensive mathematical operations can be avoided, rewrite

your code to do so. For example, if a pairwise separation distance always appears with an even power, you never need to take the square root of it to find the absolute distance.

In Fortran, rewrite mathematical formulae to result in the fewest computations possible. For example, you can break down polynomial expressions so that fewer multiplications are needed. `x**3+x**2+x+1` requires four multiplication and three addition operations. Alternatively, `x*(x*(x+1)+1)+1` requires only two multiplication and three addition operations, but gives the same result. In a similar manner, `x**8` can be evaluated fastest by `((x**2)**2)**2`.

In Fortran, precompute any groups of lumped constants. If terms are used multiple times, store their values in temporary variables so that they don't need to be recomputed.

If the same elements of a large array are to be accessed many times in succession during a Fortran loop, **copy these values into a temporary variable** first. Fortran will be able to read and write values in variables or smaller, single-dimensional arrays much faster than in large arrays because memory access can be slow and small variables can be optimized to sit in faster parts of memory. As an example, in a pairwise loop over atoms i and j, the positions of i can be stored in a temporary variable during the loop over j.

In Fortran, arrays are traversed most efficiently in memory if the **leftmost array index varies the fastest.** For example, a double loop over `Pos(i,j)` is the fastest if `i` is the inner loop and `j` the outer. Similarly, expressions like `Pos(:,j)` are faster than `Pos(j,:)`. Unfortunately, the opposite is true for NumPy and Python. Thus, it often becomes impossible to achieve the desired order of array traversal in Fortran.

In Python, **always use NumPy and SciPy routines where possible**, which are compiled and perform much, much faster than pure Python code. In particular, use an array routine over a hand-coded loop if possible. For example, consider finding the centroid (mean position) of an (N,3) array of positions called Pos. The NumPy expression Pos.mean(axis=0) is far faster than custom-writing a loop over each position, and also a bit clearer.

If absolutely necessary, very expensive mathematical expressions can be **approximated with polynomial or spline fits** to tabulated values; however, this should be done with care.

Neighbor lists can be used to speed the pairwise interaction loop. These lists keep track of all of the neighboring atoms surrounding a central atom within the cutoff distance plus some buffer distance. These lists are not recomputed at every simulation step, but only at fixed numbers of steps for which it is improbable that the neighbors will change significantly. In this way, the computation of all of the pairwise distances at each step can be avoided.

Avoid frequent writing to the screen and to files. Sending data to the screen or disk at every simulation step will greatly reduce program speed. Do these at periodic time intervals instead, spaced multiple simulation steps apart (see discussion on correlation times below).

Only compute forces if necessary. Monte Carlo and line search algorithms only require energies, so there is no reason to compute the forces as well. Accessing force arrays in memory can add a performance hit.

Techniques for making things clear

Use descriptive variable names. Python and Fortran both allow fairly long variable names. For example, Atom1Num is perhaps more informative than a1n. For variables that are used frequently, you will probably want to have a fairly short name, but often these can still be chosen to be informative.

Determine a consistent convention for names of variables and functions. If Pos denotes the (N,3) position array, then Pos_i might be the (3) vector position for one particle and Pos_ix might be that for the x-component. A common practice for functions that return Boolean values is to choose a name that starts with a verb, like IsHydrophobic or HasAtom.

It can be helpful to store intermediate values in a calculation in variables with names that describe them. This will simplify your understanding of the mathematical operations and flow of your code later on.

When speed is not an issue, **fewer commands per line is favored** for clarity. Consider:

```
Data = [map(float, l.split()[:3] for l in file(fn).readlines()
        if l.startswith("ATOM") and not l[17:20] in ["ALA", "CYS", "PHE"]]
```

The following is easier to read and debug:

```
TestSet = ["ALA", "CYS", "PHE"]
Data = []
for line in file(fn):
    Res = line[17:20]
    if line.startswith("ATOM") and not Res in TestSet:
        ThisData = [float(x) for x in line.split()[:3]]
        Data.append(ThisData)
```

Use spacing liberally to make your code clear. Use plenty of indentation. Place blank lines within functions to delineate different conceptual groups of tasks. Place multiple line breaks between functions and even more between related groups of functions.

Comment, comment, comment! Do this as you are programming. It will help you understand what your code is doing later on when you are debugging, and it will make it easier for others to

help you or to use your code. In Python, important information can be placed in docstrings for ease of access using the help function.

Keep correlation times in mind

A statistical analysis of time-series of data shows that you don't gain accuracy by including in your simulation averages samples that are more frequent than the correlation time. That is, maintaining running averages for properties evaluated at every time point is no better than waiting one correlation time to take a sample. Thus, if evaluating such averages is expensive (e.g., a detailed structural analysis), avoid taking samples frequently. You can gain an estimate of the statistical quality of your computed averages with knowledge of the correlation times.

Perform multiple trials

Multiple trials, each starting from different random velocity sets or configurations, can often generate more statistically independent samples than single long simulation runs.

Write out trajectories and restart files for long runs

For long simulations that take several days or more to perform, it is important to keep the raw data in case additional analyses are desired beyond those already planned. One should store the atomic positions at different **frames**, so as to recreate a molecular trajectory. One doesn't need to store every single configuration of the simulation, but only those configurations spaced apart by a correlation time. It can also be useful to store trajectories in compressed format, so that the corresponding files take up much less space. Python provides an easy way to do this.

It is also useful to store **restart files**, or files that are written every so often and that contain enough data (positions, velocities, etc) to restart the simulation where it left off in case of a computer malfunction or power outage.

Organize your code by level of generality

Use Python modules for large coding projects. Organize your functions by the kinds of tasks they perform, how general and reusable they are, and place related ones in common modules. Your top-level code should actually be quite short and mainly function to import and unite all of the module code you have written.

Avoid hard-coding parameters

To keep your code as general as possible and to allow you to easily tweak settings as you are debugging and executing your production runs, avoid specifying any system variables or run

parameters directly in your main code. The only numeric values that appear should be universal / mathematical constants and numbers specific to the definitions of the potential energy functions used (e.g., the coefficient 4 in the Lennard-Jones potential).

Instead, you can place all of your parameters in a file and read them at the start of a simulation. Such parameters include the system volume, number of atoms, energy potential parameters, time step, length of the simulation run, etc. You might want to have multiple files for different kinds of parameters (e.g., one specifying force field parameters and one with run length parameters).

Even better, you might keep your parameters in separate Python modules. If you have chosen descriptive variable names, this will enable a very informative file looking something like:

```
runparams.py

Volume = 100.
NAtoms = 128
Temperature = 1.2
...
```

Another advantage of a Python module over a user-defined file is that it doesn't require you to invent your own format and syntax; instead, you use Python's natural syntax, which allows comments and variable expressions.

When you have a parameter module, you can then import your parameters directly into the program namespace (e.g., so that you can use X rather than runparams.X) with the command

```
from runparams import *
```

Keep extensive system properties in extensive form

It is tempting to maintain simulation parameters in intensive, rather than extensive form. For example, you may want to specify the number density as one of the inputs to your program. This is not a good idea because ultimately simulations are of finite size and most expressions you will calculate will involve extensive variables. Moreover, the particle number and volume can fluctuate in grand-canonical and isothermal-isobaric simulations, respectively; this would make it awkward to normalize extensive properties on a per particle or volume basis.

Instead, imagine your simulation is ignorant of the concept of intensive variables as much as possible. That is, when you read your parameter file, it should specify the extensive variables, like the total number of atoms and the total volume, not the density. Moreover, when your simulation reports averages like the average energy, report the total energy and not that per particle.

Ultimately you can post-process any collected simulation data to compute intensive properties. However, simulations deal with finite-sized, small systems and therefore, the scaling of averages with system size can shed important light on the extent to which extensive behavior has been reached. More importantly, it is the extensive properties that are used to make formal connections to entropic and free-energetic quantities in many advanced simulation methods.

Keep consistent units

A common practice is to use MKS units for all variables (meters, kilograms, seconds). This will often result in very small numbers that need to be displayed in exponential notation (e.g., an oxygen mass is 2.65676×10^{-26} kg). A frequently-used alternative is "molecular" units:

length measured in Å = 10^{-10} m

mass measured in Da = 1.6605×10^{-27} kg

energy measured in kcal/mol = 6.9478×10^{-21} J / molecule

For very simple systems with one or two energy and length scales (e.g., the Lennard-Jones liquid), dimensionless units can be used. That is, the simulation units can correspond to the natural scales in the potential energy function. This approach is *not* recommended for all but the very simplest systems.

Use single arrays for most atom variables

For the fastest access and manipulation, store all atomic properties in common arrays, rather than a separate array for each atom type or molecule type. Some arrays you might have in your simulation would be:

- **(N,3) arrays** – the positions, forces, velocities, and accelerations of all of the atoms.
- **(N) arrays** – the masses (and inverse masses, for speed), charges, atom type identifiers (integers), and parent molecule numbers and/or types

The pairwise interaction loop can then detect which set of parameters to use between a pair of atoms based on their atom type and/or their parent molecule numbers/types.

Compute partial energies for moved particles

If only one atom or molecule is displaced or rotated in the simulation box, as in a typical Monte Carlo simulation, one doesn't need to recompute the total energy. Instead, one can compute the energy change for just that subset of atoms with all others, an operation that scales as N rather

than N^2. One will have to do this twice in order to update the total energy: once before moving the particle of interest, which is subtracted, and once after moving it, which is added to the current total energy.

Watch out for precision errors, part 1

Computers have finite precision that can cause problems with otherwise very normal-looking mathematical expressions. Imagine that we are performing a Monte Carlo simulation. The current energy is $U = -100$. We then make a random displacement of a particle that places it almost on top of another particle. We typically don't recompute the total energy, but instead the change in energy of that particle with all other particles. We find this energy to be $\Delta U = 10^{20}$. We update the current energy:

```
>>> U = -100.
>>> DeltaU = 1.e20
>>> U = U + DeltaU
>>> U
1e+020
```

Now, this move will invariably be rejected due to the very high energy of the new configuration. If we then want to go back to the previous energy, we might update

```
>>> U = U - DeltaU
>>> U
0.0
```

What happened? Instead of going back to $U = -100$ we now got $U = 0$. This occurs because when we add $10^{20} - 100$, there is not enough precision in the float format to contain significant figures out to the subtracted value. Thus, Python treats it as if it were still 10^{20}.

The lesson here is to always record the old value of an energy and go back to this value when doing incremental updates and rejected moves:

```
>>> U = -100.
>>> DeltaU = 1.e20
>>> OldU = U
>>> U = U + DeltaU
>>> U
1e+020
>>> U = OldU
>>> U
-100.0
```

Watch out for precision errors, part 2

Consider the computation of the following value, a typical summation that might appear in a free energy calculation

$$-F = \ln \sum_{i=1}^{M} e^{w_i}$$

In Python code we might write

```
## w is an array of M values
F = -np.log( np.sum( np.exp(w) ) )
```

This expression can fail miserably. The reason is the following, if the maximum value in the w array is even remotely large, say a value of 10000, the exponentiation will run out of precision and evaluate to infinity.

```
>>> np.exp(10000.)
1.#INF
```

This means that the entire sum over the exponential will return an infinite value and the log operation will then fail.

Instead, we can get around this numerical precision problem by rearranging our expression above so that the terms in the exponential are better-behaved:

$$\begin{aligned} -F &= \ln \left[\left(\frac{e^{w_{\max}}}{e^{w_{\max}}} \right) \sum_{i=1}^{M} e^{-w_i} \right] \\ &= w_{\max} + \ln \sum_{i=1}^{M} \frac{e^{w_i}}{e^{w_{\max}}} \\ &= w_{\max} + \ln \sum_{i=1}^{M} e^{w_i - w_{\max}} \end{aligned}$$

where

$$w_{\max} = \max w_i$$

Mathematically, this expression is identical to that above. Now, however, the exponential involves an argument whose maximum value is zero. Thus the summation must have a value of at least one. This is the proper way to perform such calculations in the computer. Our Python code now looks like:

```
wmax = w.max()
```

```
F = -wmax - np.log( np.sum( np.exp(w - wmax) ) )
```

Watch out for precision errors, part 3

One might also be interested in computing an average of the form:

$$\langle A \rangle = \frac{\sum_{i=1}^{M} A_i e^{w_i}}{\sum_{i=1}^{M} e^{w_i}}$$

This is a common reweighting scheme that we will discuss in later lectures. Here, the w_i give a measure of the weights that different configurations i should give to a particular observable A_i that can be computed for each configuration. As in the above approach, we rewrite this expression for numerical stability in the exponentiation:

$$\langle A \rangle = \frac{\sum_{i=1}^{M} A_i e^{w_i - w_{\max}}}{\sum_{i=1}^{M} e^{w_i - w_{\max}}}$$