Why “Best Practices”?

- Many scientists/mathematicians use computing in their research, but most have never taken formal Computer Science courses.

- In many fields, researchers publish computed results without ever verifying that their codes perform correctly (lack of code verification) – they just ensure that it compiles/runs and that output is “reasonable”.

- Many programmers continually “reinvent the wheel” when implementing seemingly-simple algorithms, without taking advantage of more optimal (but complex) implementations.

- Many scientific programs are designed poorly, resulting in non-optimal algorithms, messy/unclear codes, and increased development time.

- Many science/math programs require significant computation, even on modern workstations/supercomputers, so efficiency can be paramount.
1. Code Design

Thoroughly design the project up-front (before writing a line of code).

Consider the questions:
- What will the program do?
- What are the target problems?
- How should the program be used?
- How can I verify the results?

The answers to these questions can help determine many factors:
- What would be the best data structures to use?
- How should the code be organized into classes/functions?
- What are the best algorithms to use?
- And even, what is the best programming language to use?
2. Libraries

Make use of scientific software libraries whenever possible:

- These are already verified / debugged [usually].
- These are often highly optimized for each specific architecture.
- Typically include extra “bells and whistles” that you wouldn’t think of on your own.
- This is MATLAB’s approach – you construct a program by calling a number of existing optimized functions.
- Examples include: BLAS, LAPACK, FFTW, SPRNG, HDF5, MPI, ScaLAPACK, PETSc, Trilinos, HYPRE, SuperLU, ...
3. Modularity

Break off reusable portions of your code into separate functions:

- Makes it easier to reuse in your future projects.
- Makes it easier to read/understand the program.
- Makes testing/verification simpler – you can debug each component separately before trying to track down bugs in the larger program.
4. Documentation & Presentation

Thoroughly comment your code, and use spacing to help make it “pretty”:

- Eases use by others (or even yourself) later on, since the algorithms are explained in detail.

- Properly spaced code is much easier to read than compact, mangled code. Use indentation in loops, if statements, etc.; a good text editor (e.g. sublime/emacs) will do a lot of this for you.

- Code writing can be easier if you start with comments first (even including pseudocode), so that you don’t forget major steps.

- Doesn’t affect efficiency of the code at all, since the compiler strips comments and white space from the file before compilation.

- Err on the side of caution – too many comments are always better than not enough.
5. Test, Test, Test, Test, Test, Test, …

Test each portion of the program to ensure it behaves as designed.

- Use print statements everywhere to ensure that the code is doing what is expected. You can always take these out later on.

- Try out a different code with the same functionality on the same problem – do you get the same results?

- Test each algorithm with multiple, different test problems that exercise each piece of the algorithm (e.g. try a linear solver with a symmetric matrix, a random matrix, an indefinite matrix, etc.).

- This is my favorite use for Matlab/Python: although inefficient they're easy to use/debug. I like to develop/prototype my algorithms there, and then run that side-by-side with my “real” program (on a small problem) to ensure consistent results.
6. Memory Management

Keep memory management in mind as you design your algorithms.

- The order you access data can have *dramatic* performance effects.
- Programs access memory one “page” at a time – when you access a variable neighboring values are retrieved and placed into cache.
- This “page size” is system-dependent. My workstation pages hold 4096 bytes (512 doubles) – try: 
  ```bash
  $ getconf PAGESIZE
  ```
- As long as you’re getting a whole page anyway, you might as well use the data while you have it.

```c
// Bad
s = 0.0;
for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
        s += A[j*M + i];
    }
}

// Good
s = 0.0;
for (j=0; j<N; j++) {
    for (i=0; i<M; i++) {
        s += A[j*M + i];
    }
}
```
If possible, rewrite your algorithms to maximize data reuse:

```c
int i;
double u[10000];
double v[10000];
double a=0.0, b=0.0, c=0.0;
...

// no reuse
for (i=0; i<10000; i++)
    a += u[i];
for (i=0; i<10000; i++)
    b += v[i];
for (i=0; i<10000; i++)
    c += u[i]*v[i];
...
```

```c
int i;
double u[10000];
double v[10000];
double a=0.0, b=0.0, c=0.0;
...

// reuses u and v in loop
for (i=0; i<10000; i++) {
    a += u[i];
    b += v[i];
    c += u[i]*v[i];
}
...
```
7. Treat programming like a science experiment

Set up “methods” section before starting work:
- This is the design process; which subroutines/functions do you need?
- The “methods” are the parts of your program.
- You wouldn’t start mixing chemicals before weighing the ingredients, so why code without a plan?

Make “hypotheses” about what code should do on test problems:
- Determine your test problems and their expected/known results.
- Your “hypotheses” are true if you get the expected results (down to floating-point & roundoff errors).

Perform “experiments” on the code:
- Run your test problems.
- If something fails, revise your methods and re-run the experiments.

It is called “Computer Science” after all.