Summary: Best Practices for Scientific Computing

The scientific community relies heavily on software to meet the demands of their work, and it is estimated that scientists spend roughly 30% of their time creating, managing, and modifying software. *The problem is that most scientists are self-taught in software development and not exposed to basic **'good'** software development practices such as writing maintainable code, using version control, issue trackers, code reviews, unit testing, and task automation.* What follows are ten easy to utilize best practices that will make software development more efficient and manageable for scientists who develop software as a part of their regular work.

Write programs for people not computers

Break your code up into easy to understand functions that do one specific task. It's also essential that you use meaningful and consistent naming conventions throughout your code.

Here are two examples; the first is an example of what NOT to do and the second is an example of what should be done to make your code understandable to humans readers.

```
def result(x1, y1, x2, y2):
...calculation...

def rect_area(point1, point2):
...calculation...
```

Automate repetitive tasks

When coding there are numerous tasks you will do over and over such as compiling files A, B, and C together to create a resultant program. Use tools to automate repetitive tasks. A tool very often used is Make*

Use the computer to record history

Careful record keeping is fundamental to all scientific endeavors and it is essential that there is a record of data manipulations and calculations when using using software. Software can record this information automatically without additional tools:

- Unique identifiers and version numbers for raw data records
- Unique identifiers and version numbers for programs and libraries
- Values of parameters used to generate output
- Names and version numbers of programs used to generate outputs

Make incremental changes

Unlike typical software development, scientists often don't know the requirements or all steps in front of them until they try a first version of the software. Therefore, it's important to take small steps with frequent feedback to ensure progress is headed in the right direction.

Use version control

Version control will help track changes and it also allows for multiple people to work together and even simultaneously on code. In the event of conflicts, version control software will help with merging of conflicts. There are several free and for fee version control systems available including Git, Subversion, Mercurial, and VSTS

Don't repeat yourself (or others)

Modularize your code so you don't re-write the same function over and over. When it's modular, you can make updates in one place rather than searching through large code files to identify multiple duplicative updates that may be needed. It's also important to reuse libraries and packages from others to solve problems. Don't reinvent the wheel.

Plan for mistakes

Mistakes are inevitable. It's important to use defensive programming by adding assertions to validate operation. Assertions are executable documentation as well. Writing and running tests is also invaluable and includes unit tests, integration tests, and regression tests. Use Oracles to verify that your program is running or operating as it should. It's also valuable to turn bugs into test cases. Lastly, use a symbolic debugger to track down issues. No longer will you have to manually insert printf statements to find the source of a code problem.

Optimize software only after it works correctly

Don't focus on performance until your software runs and produces desired results. Only then should you focus on optimizing your code. In

fact, you may even write your code in a higher level language that requires fewer lines of code until it works properly, and, then, move it to a lower level and more performant language such as assembly.

Document design and purpose, not mechanics

It's important to document the intent behind the code and ideally inline with the code, so that when updates are made to code they are also made to documentation. It's meaningless, for the most part, to speak to the mechanics of the code as you can see here so be sure to avoid this,

$\cos(2\theta) = \cos^2\theta - \sin^2\theta$ #Computes the cosine of 2θ

Collaborate

Last but not least, it's important to collaborate with others. Have your peers and co-workers review your code. You may even consider pair programming where one person codes and the other reviews the code while it's being input into the system. There is evidence to show that pair programming positively impacts the quality of code.. . .