Reminder: Evaluations

- Please take the time to fill out the online course evaluation
- Feedback is helpful and appreciated
Safe Code Practices

- You now have some experience with version control
  - **Set up version control for your own project**
    - This pays off the first time you need to go “back in time” to figure out what recent bug messed everything up
    - Few research codes are done by a single person anymore—without version control, sharing is messy
    - Any version control system is better than none
Reuse Your Code

- You've probably seen a number of instances when doing your homework where you built from existing code
  - Not only is this much easier/quicker, it is safer
  - You debug and maintain a single routine (e.g. in a library), and all of your projects can use it. This makes testing easier
- We've written a number of Ex: matrix routines
Test Test Test

- Testing takes time
  - Test as you go along
  - Write scripts to automate it as much as possible
- Unit testing
  - For each major piece of your algorithm, think about how you might test it in isolation from the rest of the code
- Regression Testing
  - Test your output looking for changes in the results frequently
  - Together with version control, this allows you to catch and recover from recently introduced bugs
What Have We Learned

- One of the main goals of this class is that we should know what the packages we are using for our analysis, numerical simulation, etc. are doing
  - This does not mean that we should always reinvent the wheel and code everything on our own from scratch
    - Actually, that's usually a bad idea
  - Understanding the basics of these algorithms helps you understand their limitations, assumptions, and failure modes
What Have We Learned?

- **Basics of programming**
  - We got a feel for how the computer is storing the numbers we feed it
  - In most of our algorithms, we dealt with both roundoff and truncation error
  - Roundoff is mostly unavoidable (we discussed a few algorithmic tricks to minimize it)
  - For truncation error, a powerful test of your code is to check if your error converges at the expected rate
What Have We Learned?

- **Differentiation/Integration**
  - We saw different techniques for when you have $f(x)$ defined continuously vs. discretely
  - This is where we first really explored truncation error
  - Some neat methods: Gaussian quadrature

- **Interpolation**
  - Explored linear, polynomial interpolation, splines
  - Some nice examples of how higher-order is not always better
  - Trade-offs between accuracy and smoothness

- **Root-finding**
  - Newton's method came up several times in later lectures
  - Very powerful tool. We saw a few examples of what happens if your initial guess is not close enough though...
What Have We Learned?

- **ODEs**
  - $4^{th}$ order Runge-Kutta is robust. It can serve most of your needs well.
  - Stiff systems require implicit methods for accuracy
  - You should always use some form of error estimation / adaptive stepping—otherwise you have no measure of your integration error

- **Linear algebra**
  - This is a huge topic
  - Linear systems underlie many of the topics that follow
  - We looked at direct and iterative solvers
  - Lots of great libraries exist that take advantage of sparsity or symmetries in your matrices
What Have We Learned?

- **FFTs**
  - Great example of how reorganizing the computations can result in a much more efficient algorithm (still does the same computation)
  - Also a great example of a method where the publicly available routines are the way to go: they are fast, well tested, and flexible

- **Fitting**
  - General linear least squares gives us a linear system that we can solve with the matrix techniques that are widely available
  - General nonlinear least squares is much more difficult
    - Likely that some linearization is done, with N-R to solve
    - Good initial guesses are critical
    - Publicly available packages are the way to go—they are robust and tested
What Have We Learned?

- **PDEs:**
  - We saw different techniques for hyperbolic, elliptic, and parabolic PDEs
  - Many many different types of discretizations exist—we focused on finite-volume and cell-centered finite-difference
    - Different strengths and weaknesses
  - We discussed a bit about how to deal with scalar equations and systems that mix the types of PDEs
  - The Euler equations was our introduction on how to deal with nonlinear hyperbolic systems

- **Parallel programming:**
  - OpenMP for shared-memory—easy to get started, can do piece-by-piece
  - MPI for distributed—requires you to rethink the algorithm and layout of the data
What Didn't We Cover

- **SIAM Editors: Top 10 Algorithms of the 20th Century** (SIAM News, 33, 4)
  - Monte Carlo
    - We did MC integrals and MCMC
  - Simplex method for linear programming/optimization
    - Looks to maximize: \( cx \) with \( Ax \leq b, x \geq 0 \)
    - Real problems may be nonlinear
  - Krylov subspace iteration
    - We briefly referenced one method: conjugate gradient
    - Works in the subspace of powers of \( A \): \( A^0b, Ab, A^2b, \ldots \)
    - Solve linear systems iteratively
  - Decompositional approach to matrix computations
    - E.g. LU decomposition
  - Fortran optimizing compiler
What Didn't We Cover

- Continued: SIAM Editors: Top 10 Algorithms of the 20th Century
  - QR algorithm
    - Method for computing eigenvalues
  - Quicksort algorithm
    - NlogN method for sorting numbers (very robust)
  - FFT
  - Integer relation detection algorithm
    - Given $x_1, x_2, ..., x_n$ find numbers $a_1, a_2, ..., a_n$ such that $a_1 x_1 + a_2 x_2 + ... + a_n x_n = 0$
    - Apparently useful in Feynman diagram calculations
  - Fast multipole algorithm
    - Reduces $N^2$ calculations in N-body calculation to $O(N)$ using multipole expansions and a hierarchical decomposition of the domain
What Else Didn't We Cover?

- N-body/Tree Codes/Molecular Dynamics
- SPH
  - We ran out of time for this one
  - SPH is a popular hydrodynamics method used in astrophysics
  - Instead of a grid, a collection of particles samples the mass distribution
  - Continuous quantities are formed by integrating over nearby particles with a smoothing kernel
Where Do Physics Algorithms Get Published?

- Most journals publish algorithms
  - Journal of Computational Physics is perhaps the main dedicated journal
  - ApJ publishes many code papers and new algorithms
  - SIAM journals publish more applied math stuff
- One of the difficulties you will encounter is that quite often, the little details of algorithms are left out of a paper
  - This is both for space considerations, or that it is assumed to be obvious
  - If you spend months/years writing a code and then go to write the paper, you probably forget some of the minor design decisions, trade-offs, nuances that were encountered over the development period
Supercomputing Centers

- **Supercomputing centers**
  - National centers run by NSF (through XSEDE program) and DOE (NERSC, OLCF, ALCF)
  - You can apply for time—starter accounts available at most centers to get up to speed
  - To get lots of time, you need to demonstrate that your codes can scale to $O(10^4)$ processors or more

- **Queues**
  - You submit your job to a queue, specifying the number of processors (MPI + OpenMP threads) and length of time
  - Typical queue windows are 2-24 hours
  - Job waits until resources are available
Supercomputing Centers

- Checkpoint/restart
  - Long jobs won't be able to finish in the limited queue window
  - You need to write your code so that it saves all of the data necessary to restart where it left off

- Archiving
  - Mass storage at centers is provided (usually through HPSS)
  - Typically you generate far more data than is reasonable to bring back locally—remote analysis and visualization necessary
No one likes to write documentation

Serves multiple purposes:

- For developers, it preserves the train-of-thought that lead to design decisions, serves as a reference for complicated pieces of the algorithm, helps you get back into things after some time away
- For new users, it serves as the introduction

Sharing your code? writing documentation once saves you from having to answer the same questions over and over

- Start small: document pieces as questions arise or major design changes are done. This lets you gradually build up.
I/O

- For many projects, output is not a major concern
  - If you are just outputting a few columns of numbers, then ASCII works well, is portable
  - Not very space-efficient, but typically doesn't matter with small outputs
- What if you need to write lots of data?
  - Simulation outputs of ~10 to 100 GB per file are not unusual anymore
  - You must store this in binary form
  - Binary is generally not portable across machine architectures
I/O and Endianness

- Different architectures store binary numbers with different byte orderings (endianness)
  - Big-endian: most significant byte is stored first (starting address)
    - Examples: Motorola 68k, IBM POWER
  - Little-endian: least significant byte is stored first
    - Examples: x86, DEC Alpha, VAX

<table>
<thead>
<tr>
<th>Endian</th>
<th>First byte (lowest address)</th>
<th>Middle bytes</th>
<th>Last byte (highest address)</th>
<th>Decimal 100000000 (hexadecimal 5F5E100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>big</td>
<td>most significant</td>
<td>...</td>
<td>least significant</td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td></td>
<td></td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>n+1</td>
<td></td>
<td></td>
<td>F5</td>
</tr>
<tr>
<td></td>
<td>n+2</td>
<td></td>
<td></td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td>n+3</td>
<td></td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>little</td>
<td>least significant</td>
<td>...</td>
<td>most significant</td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td></td>
<td></td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>n+1</td>
<td></td>
<td></td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td>n+2</td>
<td></td>
<td></td>
<td>F5</td>
</tr>
<tr>
<td></td>
<td>n+3</td>
<td></td>
<td></td>
<td>05</td>
</tr>
</tbody>
</table>

(Wikipedia)
I/O and Endianness

- It is most efficient for the machine to write out in its native format.
- On the opposite endian machine, you will need to manually swap bytes as they are read in (can be slow).

Some file formats / libraries handle this automatically

- HDF5: transparently handles endian differences, supports parallel I/O, self-documenting
- NetCDF: similar to HDF5
- FITS: standard doesn't specify endianness, but NASA apparently says big-endian

- If you are going to make data publically available, using a standard I/O format/container may make things easier.
Sharing Codes

- Sharing codes is good
- Some things to think about:
  - Documentation
  - Support
  - Licensing
Final Thoughts...

- If you find yourself doing the same thing over and over, automate it
- Ask around