Numerical Software Engineering 101/201

Scientific Software Club 2/13/17
Papers

- *Barely Sufficient Software Engineering: 10 Practices to Improve your CSE Software*, Heroux and Willenbring
Misconception: Coding is unimportant! It’s not like I’m a software engineer...

(The crucial part is getting the numerical algorithm, proper data, good results, etc)
The (Relative) Truth: Coding is an important part of research and a skill that takes years to hone

*(Teach Yourself Programming in Ten Years by Peter Norvig)*
Topics

- Code Level Management
- Data Management
- Directory Level Management
- Project Level Management
- Working with Others
- Documentation and Technical Writing
Code Level Management
double AreaRectangle(double x, double y){
    /* AreaRectangle calculates the area of a rectangle with dimensions x and y */
    /* Return -1 if bad input*/
    if(x < 0 || y < 0){
        printf("x and y must be positive numbers");
        return -1;
    }
    /* Return the product of x and y */
    return x*y;
}
/*
runN4SID runs the system identification algorithm n4sid

~~~~~INPUT~~~~
data: N x K time domain signal, N = number samples, K = dimension of data
p: includes measurement frequency in Hz, model size to fit
~~~~~~~~~~~~~

~~~OUTPUT~~~
Fitted system model, saved in results folder as system.csv
~~~~~~~~~~~~~
*/
void runN4SID(double data, params p){
    ...
}
}
Name Intelligently

- Fits in with earlier example, but having descriptive function and variables is extremely important
- A headache for numerical calculations
  - Generally, code might be ugly, but make sure function is named well!
void calcStuff(...){
    A = getMatrix(...);
    [U, D, V] = svd(A);
    [X, Y] = getData(...);
    [E, Z] = eig(X*A*Y);
    w = getWeights...();
    [S, N] = sumEV(W, w);
    B = convolveMatrix(A, N, S)
    I = [ identity(N); identity(N)];
    C = I*B + I*A;
    [Q, R, P] = qr(C);
    ....
    (you get the point)
}
class Central2D {

public:

    Central2D(float w, float h, int nx, int ny, float cfl = 0.45) : // Domain width / height
        cfl(cfl) {

        nx(nx), ny(ny),
        nx_all(nx + 2*nghost),
        ny_all(ny + 2*nghost),
        dx(w/nx), dy(h/ny),
        // Array accessor functions
        nx_all, ny_all;

        const int nx, ny; // Number of cells in x/y (without ghosts)
        const int nx_all, ny_all; // Total cells in x/y (including ghost)
        const float dx, dy; // Cell size in x/y
        const float cfl; // Allowed CFL number

        // Array accessor functions
        int offset(int ix, int iy) const { return iy*nx_all+ix; }

        float& u1(int ix, int iy) { return u1_[offset(ix,iy)]; } // Solution values
        float& u2(int ix, int iy) { return u2_[offset(ix,iy)]; }
        float& u3(int ix, int iy) { return u3_[offset(ix,iy)]; }
        float& f1(int ix, int iy) { return f1_[offset(ix,iy)]; } // Fluxes in x
        float& f2(int ix, int iy) { return f2_[offset(ix,iy)]; }
        float& f3(int ix, int iy) { return f3_[offset(ix,iy)]; }
        float& g1(int ix, int iy) { return g1_[offset(ix,iy)]; } // Fluxes in y
        float& g2(int ix, int iy) { return g2_[offset(ix,iy)]; }
        float& g3(int ix, int iy) { return g3_[offset(ix,iy)]; }
        float& ux1(int ix, int iy) { return ux1_[offset(ix,iy)]; } // x differences of u
        float& ux2(int ix, int iy) { return ux2_[offset(ix,iy)]; }
        float& ux3(int ix, int iy) { return ux3_[offset(ix,iy)]; }
        float& uy1(int ix, int iy) { return uy1_[offset(ix,iy)]; } // y differences of u
        float& uy2(int ix, int iy) { return uy2_[offset(ix,iy)]; }
        float& uy3(int ix, int iy) { return uy3_[offset(ix,iy)]; }
        float& fx1(int ix, int iy) { return fx1_[offset(ix,iy)]; } // x differences of f
        float& fx2(int ix, int iy) { return fx2_[offset(ix,iy)]; }
        float& fx3(int ix, int iy) { return fx3_[offset(ix,iy)]; }

        // Diagnostics
        void solution_check();
        // Array size accessor
        int xsiz() const { return nx; }
        int ysize() const { return ny; }

        // Read / write elements of simulation state
        float& operator()(int i, int j) {
            return u1_[offset(i,j)];
        }

        const float& operator()(int i, int j) const {
            return u1_[offset(i,j)];
        }

        // Wrapped accessor (periodic BC)
        int ioffset(int ix, int iy) {
            return offset( (ix+nx-nghost) % nx + nghost,
                           (iy+ny-nghost) % ny + nghost );
        }

        float& uwrap1(int ix, int iy) { return u1_[ioffset(ix,iy)]; }
        float& uwrap2(int ix, int iy) { return u2_[ioffset(ix,iy)]; }
        float& uwrap3(int ix, int iy) { return u3_[ioffset(ix,iy)]; }

        // Call f(Uxy, x, y) at each cell center to set initial conditions
        void run(float tfinal);
    }
};
Decompose Programs into Functions

- Try to keep functions short
- Modularity makes code base more flexible, more easily modifiable
- Saves lines of code
- Practically speaking, humans can only remember a few things at a time!
void calcStuff(...){
    Node root;
    ...
    Node data;
    ...
    bool checkchild = 0;
    for(i = 0; i < root.numchildren; i++){
        if(root.child[i] == data){
            checkchild = 1;
        }
    }
    ...
    
}

void calcStuff(...){
    Node root;
    ...
    Node data;
    ...
    bool checkchild = isChild(root, data);
    ...
    
}
Eliminate Duplication

double calcValues(...){
    ...
    X = getValue(...);
    return X;
}

double calcValuesFilter(...){
    ...
    X = getValue(...);
    X = filter(X);
    return X;
}

double calcValues(..., bool Filter){
    ...
    X = getValue(...);
    if( Filter == true){
        X = filter(X);
    }
    return x;
}

VS
Keep Semantics Consistent

```c
void scaleVec(vec v, double n){
    ...
}

void filterEigenVecs(Matrix M){
    ...
}

void find_all_keys(keys K){
    ...
}

VS

void scaleMatrix(double n, matrix m){
    ...
}

void filterEigVals(Matrix M){
    ...
}

void findAllKeyrings(rings R){
    ...
}
```
Use Data Structures (If necessary)

```cpp
void doStuff(...
double timestep, int size...
date d, int dimx, int dimy...
int numthreads){
...
}

VS

void doStuff(metatdata d){
    ...
}

class metadata{
    double timestep;
    int size;
    date d;
    int dimx;
    int dimy;
    int numthreads;
}
```
Incremental Changes

- Emphasized in two papers
  - Decompose a large task into small components
  - Test the correctness of components

- Programmers are most productive working in small steps
  - + Course Correction
Defensive Programming

- Assert (or Try/Catch)
- Unit Testing
  - What if no “useful” unit tests?
  - Numeric Unit Tests
- Automated Testing and Continuous Integration
  - (to be covered in the future)
Abstractions

- Computer Systems Researchers often talk about getting the right “abstractions”
  - “Abstraction” decrease the complexity of your software by making the low-level details hidden from the user
- Defining a convenient way to interact with your code base is hard!
  - Takes practice... cannot be quantified
  - What do you expose to the user (one of which will surely be yourself)?
Data Level Management
Save Raw and Intermediate Data

- Raw data D >> Intermediate Forms >> Result (yes or no)
  - You don’t just want to save the yes/no!
- Save Raw and Intermediate Forms
  - Saves time, extra processing, etc
Format Data Well

- Create data you wish to see in the world
  - Neatly labeled columns, information on format, etc
  - Important, especially if your data format changes down the road

- Space is cheap!
  - One variable per column, one observation per row, etc
  - Don’t cram!
Manage Your Metadata

- What is “Metadata”?  
  - In short: Data about Data Set
- Might include date produced, units, etc
- You’ll need it later!
Publish Data

- (If you think others might want to use it)
- “Your data is as much a product of your research as the papers you write”
- Figshare, Dryad, Zenodo
Directory Level Management
Directory Names

- Your project should NOT be named “foo” or “a”
- Subdirectories should also be descriptive
  - Documentation in “docs”
  - Source in “src”
  - Scripts in “bin”
  - Etc...
- Should include a “data” and “results” folder
  - Make a distinction between what goes in each folder, as your results will surely contain data!
  - Idea: every output goes in “results”, every input goes in “data”
Directory Names

- README
- LICENSE
- Tests
  - testSightings.py
- data
  - birdcount.csv
- doc
  - notebook.md
  - changelog.txt
- results
  - summarized-results.csv
- src
  - Sightings.py
Subdirectories (Don’t make too many)

- src
  - helpers
    - datastructs
    - graph
      - graphsearch
      - methods
        - dfs.py
Don’t Repeat Previous Work

- Use external libraries as much as possible
  - Optimized code and saves development time
- Use google, github, cppreference, etc
Project Level Management
Version Control

- Discussed Earlier This Semester
- Git, CVS, Mercurial, etc
  - Git preferred (Github, Bitbucket)
- Commit often, Commit early
- Don’t add large data dumps/files!
  - Makes version control slow, impractical
  - We will discuss later in semester how to manage this stuff
Adding Features, Refactoring

- Add features incrementally
  - Constantly check correctness
  - Don’t expect to add 1k+ lines and have your code work the first time
- Refactoring is a natural part of coding
  - Don’t avoid it
  - End up with bloated code
To use an IDE or not to use an IDE...

- I’m not sure!
  - What if like Microsoft Visual Studio, Eclipse, PyCharm?
  - Problem: code should be accessible to everyone
  - Getting libraries integrated into an IDE can be painful
    - For numeric libraries, even more annoying
    - Software makes this easier e.g. Intel Parallel Studio XE, Nividia NSIGHT, etc
  - If you’re prototyping and know IDE’s debugging and profiling tools well, why not
  - Mismatch between IDE environment and deployment environment
Issue-Tracking Software

- Common Mistake
  - “I need to refactor A, B, C and debug I, J, K
  - (One seminar and one nap later) “What was I supposed to do again?”

- Many out there (Wikipedia lists ~ 50)
  - Bugzilla, Apache Bloodhound, Planbox, etc etc
### Active Tickets by Version

This report shows how to filter results by priority while grouping results by version.

Last modification time, description and reporter are included as hidden fields for useful RSST support.

<table>
<thead>
<tr>
<th>Ticket</th>
<th>Summary</th>
<th>Component</th>
<th>Version</th>
<th>Type</th>
<th>Owner</th>
<th>Status</th>
<th>Created</th>
</tr>
</thead>
<tbody>
<tr>
<td>141</td>
<td>Install tabs when the app is down</td>
<td>installer</td>
<td>defect</td>
<td></td>
<td>gzf</td>
<td>assigned</td>
<td>Aug 12, 2012</td>
</tr>
<tr>
<td>146</td>
<td>Improve editing of objects</td>
<td>dashboard</td>
<td>enhancement</td>
<td>change</td>
<td></td>
<td>accepted</td>
<td>Jul 17, 2012</td>
</tr>
<tr>
<td>151</td>
<td>Users should be able to watch objects</td>
<td>dashboard</td>
<td>enhancement</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>Aug 13, 2012</td>
</tr>
<tr>
<td>156</td>
<td>Local copy of bloodhound part of Apache repo for browse</td>
<td>sha/output</td>
<td>task</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>Aug 7, 2012</td>
</tr>
<tr>
<td>159</td>
<td>Incorporate sha 1.1 into master branch</td>
<td>sha/core</td>
<td>task</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>Aug 11, 2012</td>
</tr>
<tr>
<td>157</td>
<td>Request that the ThemeEngine allows them to implement their own theme</td>
<td>plugins</td>
<td>task</td>
<td>nobody</td>
<td>assigned</td>
<td></td>
<td>Apr 4, 2012</td>
</tr>
<tr>
<td>153</td>
<td>Update styling for login page to be responsive</td>
<td>dashboard</td>
<td>task</td>
<td>obsecure</td>
<td>accepted</td>
<td></td>
<td>May 1, 2012</td>
</tr>
<tr>
<td>154</td>
<td>Exercise workflow for Bloodhound #1</td>
<td>sha/admin</td>
<td>task</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>Aug 21, 2012</td>
</tr>
<tr>
<td>161</td>
<td>Generic conversion for admin table listings and additions</td>
<td>dashboard</td>
<td>defect</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>Jun 10, 2012</td>
</tr>
<tr>
<td>151</td>
<td>Make areas for reply and follow-up comments point to correct direction on ordering change</td>
<td>dashboard</td>
<td>defect</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>Aug 28, 2012</td>
</tr>
<tr>
<td>147</td>
<td>Investigate enhanced selection of active tickets in mobile</td>
<td>dashboard</td>
<td>enhancement</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>Apr 30, 2012</td>
</tr>
<tr>
<td>156</td>
<td>Split QuickSearchTicketDialog for more flexible approach</td>
<td>dashboard</td>
<td>enhancement</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>May 2, 2012</td>
</tr>
<tr>
<td>150</td>
<td>Alert text widget in quick ticket</td>
<td>dashboard</td>
<td>enhancement</td>
<td>nobody</td>
<td>now</td>
<td></td>
<td>May 3, 2012</td>
</tr>
<tr>
<td>167</td>
<td>Highlight tickets submitted by reports</td>
<td>az/design</td>
<td>enhancement</td>
<td>change</td>
<td>accepted</td>
<td></td>
<td>May 14, 2012</td>
</tr>
</tbody>
</table>
Working with Others
Industry vs Academia

- In industry, a group of experienced engineers is often assigned to manage a single piece of software
- In academia, a single person might manage multiple pieces of software
Getting a Second Look

- Just as research ideas need a second look, so does a potential code base.
- Pair Programming is extremely beneficial
  - Could be a problem if you’re the only one working on a project.
- Coding with others ultimately makes you a better programmer.
Documentation and Technical Writing
Create Barely Sufficient Documentation

- Somewhat covered earlier last semester
  - Documentation generation via Sphinx, Doxygen, etc
- You are writing the documentation for yourself as well as others!
Document All Work You’ve Done

- Not just the code you plan to release; code you’ve written but not used, ideas you’ve tried (both successful and unsuccessful), etc
Reports and Papers

- Writing a paper or technical report? Put it under version control as well
- Formal Approach: Treat paper/report writing as programming.
- Save you time and effort town the road
Figures

- One script per figure
- Don’t manually change parameters; input them into functions
- Automation
  - Don’t be tempted to manually adjust window size and click the “save as” button in MATLAB
Conclusions
Conclusions: Takeaways

- Following software engineering best practices saves development time, headaches, and user-friendliness
- Developing (and maintaining) software is hard!
Conclusions: Questions

- Why put in all this effort if no one else is going to use my code?
- Considering the time spent improving non-essential parts of my code, will the time saved from following best practices be greater than the extra development time invested?