Coding for Researchers

January 16, 2014

Lancaster University
Peter Frazier

How I learned (then later forgot) to write good code

- I earned my undergraduate degree from Caltech in 2000, with a dual major in computer science and physics.
- As a CS student, I wrote code in various languages (C, C++, Java, Lisp, UNIX shell scripting) for:
  - class assignments
  - one semester long course project
  - summer jobs
  - projects for fun

How I learned (then later forgot) to write good code

- Upon graduation, I joined a small startup company founded by a friend of mine.
- I mostly developed a web client in PHP, with bits of C and Perl along the way.
- I learned that working in a group on a big project is different than working individually on small projects.

How I learned (then later forgot) to write good code

- Our startup eventually folded, and I joined another startup.
- I wrote code in C for the backend of a relational database.
- My code was reviewed by senior programmers with excellent skills, who told me how to improve my code.
- This is where I really learned how to code.

How I learned (then later forgot) to write good code

- This startup eventually laid most people off, and I joined a big company.
- I wrote code in C for the filesystem of a proprietary UNIX maintained by a relational database company.
- I learned that writing code at a big company is very different from writing code at a small company.

How I learned (then later forgot) to write good code

- I got bored, and decided to go get a masters & PhD in Operations Research at Princeton.
- I was a student from 2005 until 2009, and wrote code in Matlab and Java for my dissertation research, along with bits of C and Java.
- I learned that writing code for research as a graduate student is very different than writing code for customers as a software engineer.

How I learned (then later forgot) to write good code

- In 2009, I graduated and became an assistant professor at Cornell's Operations Research department.
- I stopped writing lots of code myself, and started supervising students who write lots of the code.
- I learned that supervising students who write code is very different than writing code yourself.
- My coding skills have slowly declined over the years, but I hope to still tell you a few useful things...

Is it worth your time to improve your coding?

- In this talk I will make some suggestions about things you can do to improve your coding.
- Acting on any one of these suggestions would require time & energy.
- This is time & energy that you could instead be using to “be productive”.
- You will have to decide whether it is worth it.

Improving your coding has these advantages

- 1. You will be able to do research more quickly.
- 2a. Your code will be easier for others to understand and use.
- 2b. This will make others more likely to use your code.
- 2c. This will increase the impact of your research.
- 3. Coding well may help you get a job, and may make you more effective in that job after you get it.

Specific Advice

- Learn from people who code well
- Split your code into functions
- Create unit tests
- Document your code
- Add error handling
- Use a good text editor
- Use version control
- Save the input & output to expensive experiments
- Save the raw data you use to generate plots
- Use profiling to observe which parts of your program contribute most to runtime.
- Understand a bit about computer architecture.
- Use system monitoring tools to observe CPU, disk and network to find the bottleneck.

Learn from people who code well

- If your office mates code well, ask them questions.
- If your office mates code badly, help them.
- If you have the opportunity:
  - Look at well-written code.
  - Encourage a good coder to review your code & make suggestions.

Split your code into functions

- Each function should do a specific task, transforming some well-defined inputs into some well-defined outputs, in a well-defined way.
- Aim for no more than 200 lines of code in a function.
- If a function is longer, ask whether it can be broken into smaller functions.

Split your code into functions

- "A program should not require its readers to hold more than a handful of facts in memory at once"
- Breaking code into functions reduces the number of things you need to remember when writing code.
- This reduces bugs, and allows you to write code faster.

Split your code into functions

- Splitting your code into functions makes it easier for others to understand your code.
- It also makes it easier for you & others to reuse your code.

Create unit tests

- A unit test is a piece of code that tests a single function or unit of your code.
- They are worth the work because it is easier to fix bugs if you find them early.
- Imagine being close to submitting a paper, or finishing a chapter of your dissertation, and discovering that your main numerical result is invalid due to a bug.
- Find bugs early!

Here are some example unit tests

- For code that solves a dynamic program, simulate the optimal policy and compare its estimated value to what the solution code claims it should be.
- For code that estimates something from data, generate lots of fake data where you know the something and see if your code can estimate it correctly.

Here are some more unit tests

- If you wrote some old code that computes something in a special case, and new code that computes it more generally, run both codes on random instances where both should work, and compare the output.
- If you wrote old slow code and new fast code that compute the same thing, run them both on random instances and compare the output.
- Run generic code on random inputs and see if it crashes.

Turn bugs into unit tests

- If a bug arises outside of a unit test:
  - Write a new unit test that reproduces the condition under which that bug arose, and checks whether that bug is still happening.

Here's how I use unit tests

- I make my unit tests functions test1(), test2(), etc. that take no arguments, print out a message "OK" or "FAIL", and return a single bit.
- I then make another function test() that runs all the unit tests, one after the other, and reports whether everything passed or not.
- This makes it easy to test whether everything still works after a software upgrade, or on a new system.

Document your code

- Describe inputs & outputs at the beginning of your functions.
- Use variable and function names that mean something (without being too long).
- If you have a lot of files, separate them into directories that make sense, and add a README.txt file in each.
- This is useful because:
  - it makes it easier for others to read your code
  - it makes it easier for you to read your code, several months down the road when you've forgotten what you were thinking when you wrote it.
  - it makes you think about the structure of your code.

Here is some example documented code

```
%[impl,x,y,extras] = SimAdaptiveCKG(N,sample,dim,n0a,n0b,MPerDim,MLE)
%
% Simulates CKG algorithm through N samples, and collects at each sample time
% 1<=n<=N the measurement and implementation decision.  Uses the passed
% MLE to estimate hyperparameters, including the noise variance.
%
% N -- number of measurements to take.
% sample(x) -- function that, given a (flattened) measurement location x, returns a
%   noisy sample of the function value at that point.
% dim -- number of dimensions in the search domain.
% n0a -- number of measurements to put in the first stage latin hypercube.
% n0b -- number of measurements to put in the second stage, when we measure
%       again the n0b best points.
% MPerDim -- number of discretizations in each dimension.  A scalar.
% MLE -- function handle of the form,
%       [cov0hat, noisevarhat, scalehat, betahat] = MLE(xd, y)
%
% impl -- vector of implementation decisions
% x -- vector of measurement decisions
% y -- vector of observations
% extras.cov0 -- array of estimated cov0
% extras.mu0 -- array of estimated mu0
% extras.scale -- array of estimated scale
% extras.noisevar -- array of estimated noise variances
% extras.logkgfactor -- array of max(logkgfactors)
%
function [impl,x,y,extras] = SimAdaptiveCKG(N,sample,dim,n0a,n0b,MPerDim,MLE)
    M = MPerDim^dim;
    impl = zeros(1,N); % preallocate for speed.
    x = zeros(1,N);
    y = zeros(1,N);
    extras.cov0 = NaN*ones(1,N);
    extras.mu0 = NaN*ones(1,N);
    extras.scale = NaN*ones(dim,N);
    extras.noisevar = NaN*ones(1,N);
    extras.logkgfactor = NaN*ones(1,N);

    toZd = @(z) ZToZd(z,MPerDim,dim);

    % Sample n0a points using latin hypercube sampling.
    n0 = n0a+n0b;
    assert(n0a>0);
    xd(1:n0a,:) = DiscreteLHS(MPerDim,dim,n0a);
```

Documenting your code is useful

- Documenting your code is useful because:
  - it makes it easier for others to read your code
  - it makes it easier for you to read your code, several months down the road when you've forgotten what you were thinking when you wrote it.
  - it makes you think about the structure of your code.

Documenting your code is useful some of the time

- Sometimes we write code that we are just going to throw away right afterward.
- If you really will throw it away, it is ok not to document it.
- It is often hard to know, when you write it, whether you will throw it away.
- If you find yourself continuing to use old code that is undocumented, take the time to clean it up & document it.

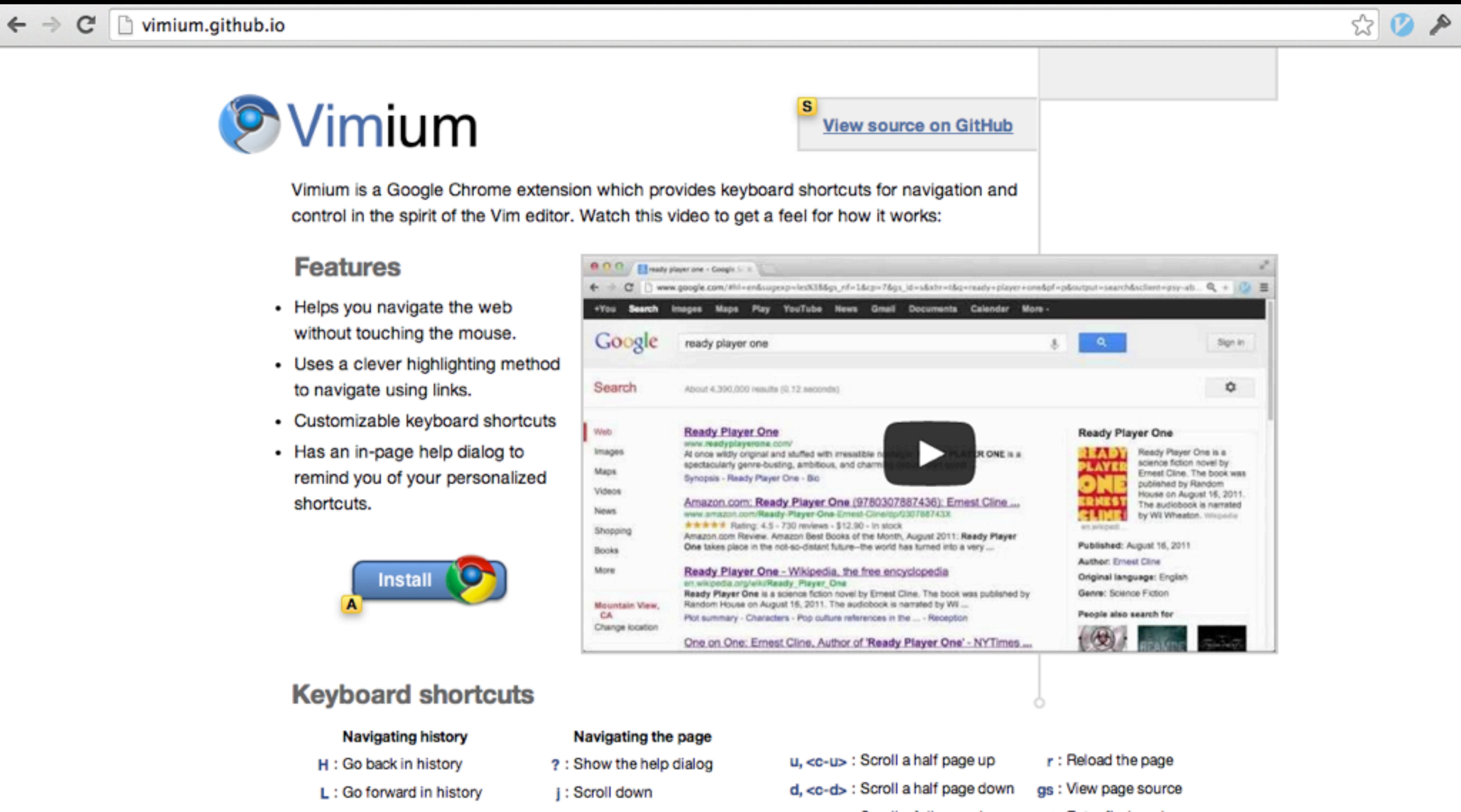
Add error handling

- If a function assumes something about its inputs, add code that checks whether these assumptions are met.
- If they are not, print out an error message and return from the function.
- This makes it easier for you to fix it when your code crashes.
- This makes your code catch errors that would have silently caused incorrect output.
- This makes your code more usable by others.

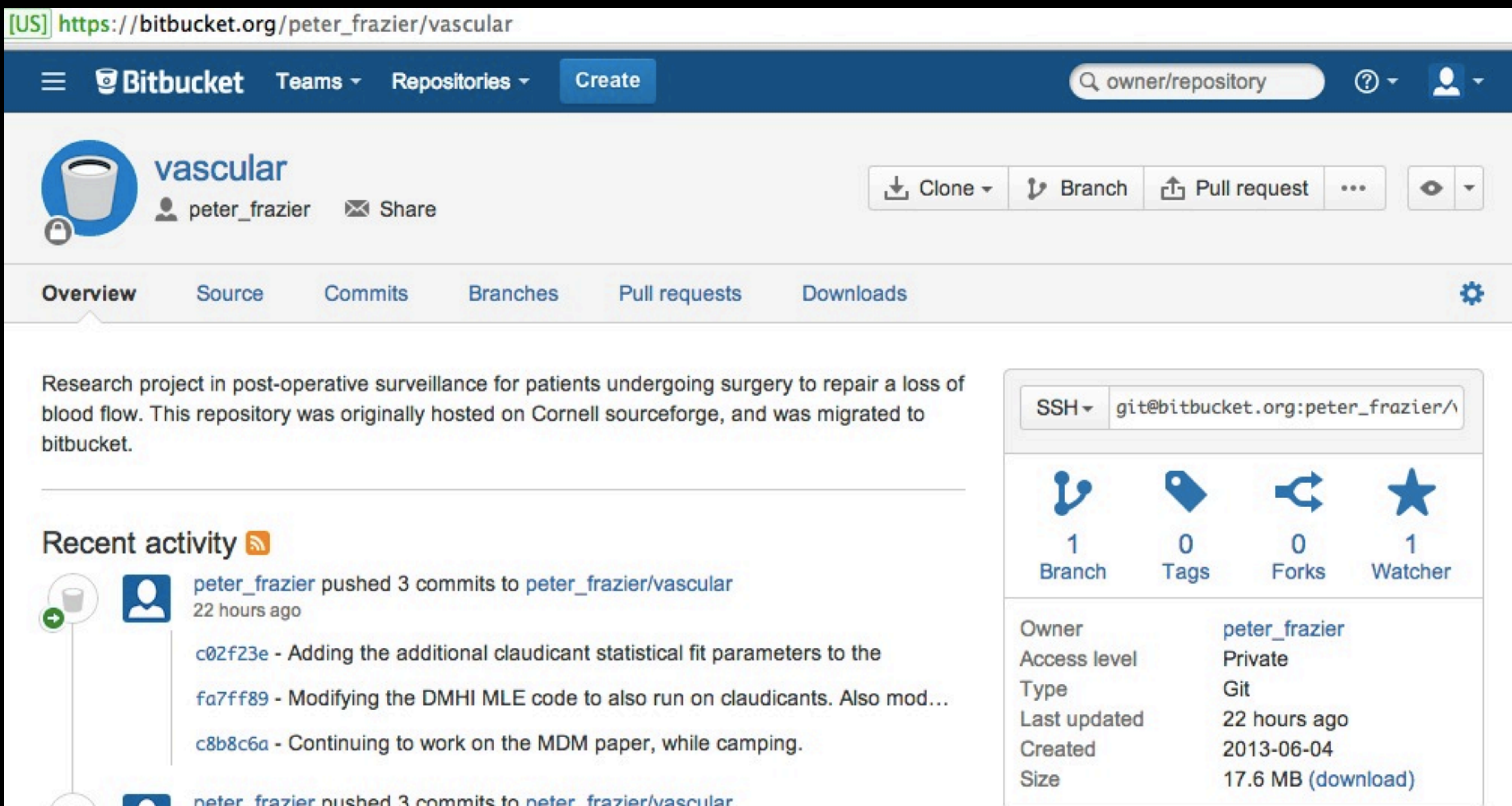
Use a good text editor

- I am aware of two reasonable choices:
  - vi / vim
  - emacs
- It should be possible to move the cursor anywhere on the screen, or cut & paste from any location to any other location, with 5 keystrokes or less (and no mouse), most of the time.

If you use vim, also try the Vimium chrome extension



Use version control (git w/ bitbucket or github)



There is a nice tutorial on the bitbucket website

Save the output from expensive experiments

- Suppose your computational experiments take several hours to run.
- Have your code output & save **everything** you might reasonably be interested in the future, subject to space constraints.
- For example, if you are calculating mean & variance using Monte Carlo, save each replicate.
- Have your code save its inputs & outputs to a file.
  - In Matlab, .mat format is convenient.
  - In many languages (R, C/C++, Matlab), .csv is convenient and portable.
  - In object-oriented languages (C++, Python, Java), you can create classes that are serializable, which allows writing and reading complex objects directly to files.

Saving output from expensive experiments has advantages

- Separate your post-processing code by having it read from these stored data files, rather than directly from the expensive experiment code.
- Later, if you want to perform some other analysis on the data, or fix a bug in your post-processing code, you don't need to re-run the experiment.
- You can write your post-processing code in a different language than the expensive experiment code.
- For example, write your expensive experiment in C, and your post-processing in R.

Save the input to expensive experiments

- Several months later, when you come back to a saved output file, you won't remember the inputs or version of the code that you used to generate it.
- Have your experimental code print out & save:
  - all inputs
  - the current date and time
  - a version number for the code.

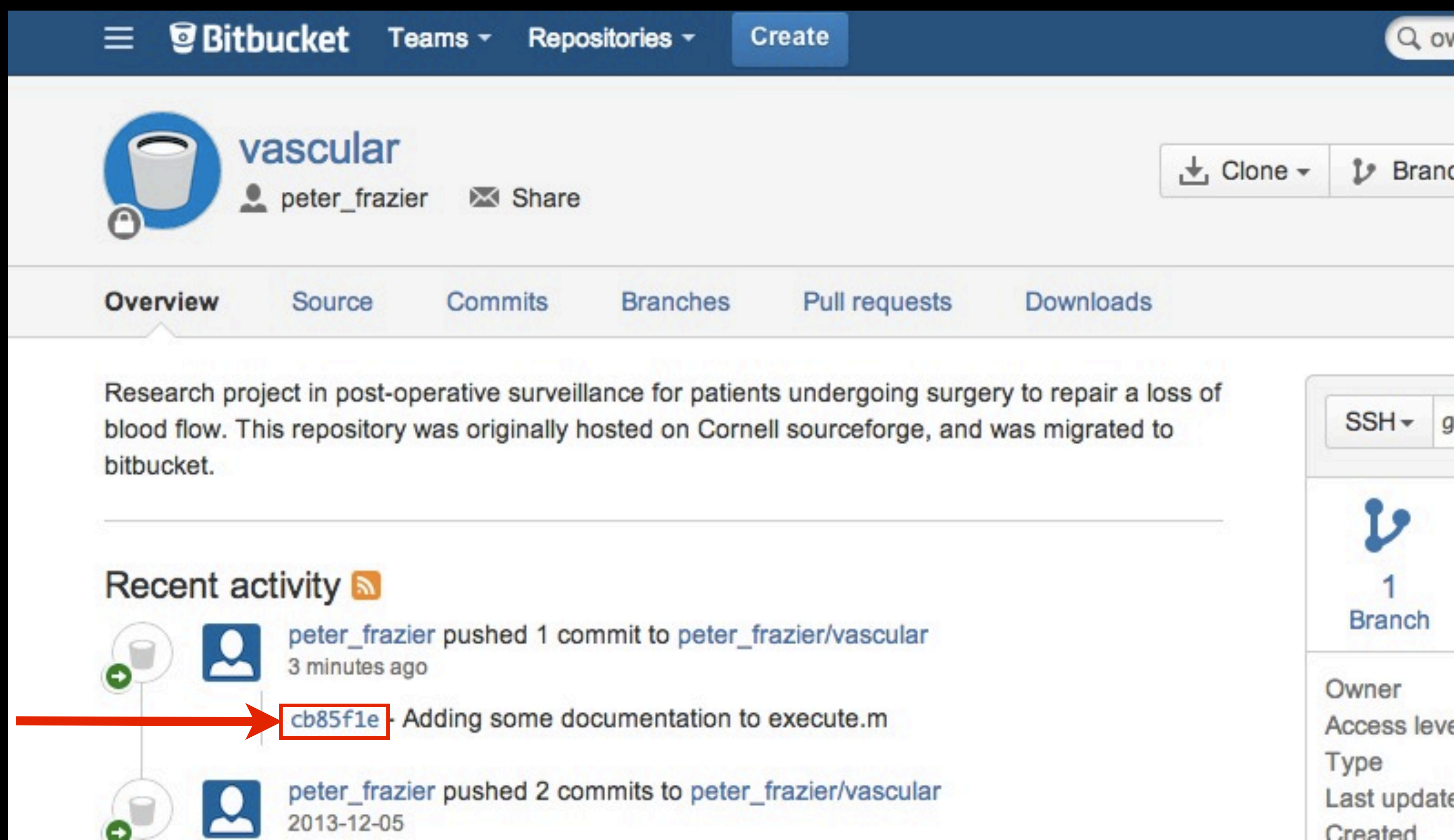
If you use git, here is a nice way to get a "version number"

- Run "git rev-parse HEAD"
- This returns the hash (unique identifier) of the most recent commit.
- You can then look up this commit later in git's version history.

If you use git, here is a nice way to get a "version number"

I ran this on the command line:

```
en-or-apf98-a-2% git rev-parse HEAD
cb85f1e8687de718ae41d8f4e9fadb3970d508b3
```



I can then compare the first 7 characters to the website:

It might also be a good idea to save the output of “git status”

- In theory, one would always commit all of your changes before running an expensive experiment (keep in mind that you can use a branch if this is just a “throw away” experiment).
- In practice, one might forget to do this.
- Saving the output of “git status” will let you know whether you have any uncommitted changes.

Here is some example output from “git status”

```
en-or-apf98-a-2% git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   ../presentations/2013.06.INFORMS_Healthcare/2013.06.INFORMS_Healthcare_Frazier_Vascular.key
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       ../code/opt/my_fatality.m
#       ../code/opt/my_fminconop.m
#       ../code/stat/Test_stats.m
#       ../data/Copy of WCMC_LEDBS_safe.xlsx
#       ../data/Copy of WCMC_LEDBS_safe9-27_2.xlsx
#       ../data/ResearchData.xlsx
#       ../data/WCMC_LEDBS_safe.xlsx
#       ../data/WCMC_LEDBS_safe10-18.xlsx
#       ../data/WCMC_LEDBS_safe10-18_plus_cols_AP_AQ.xlsx
#       analysis/.plotOptimalVsEndTime.m.swp
#       stat/.Oct18_erlang_exp_likelihood.m.swp
#       ../mdm_paper/MDM_paper_AJM.docx
no changes added to commit (use "git add" and/or "git commit -a")
```

Similar things can be done with subversion

- Use "svn info" to get the current revision.
- Use "svn status" to get a list of modified files.

Save the raw data you use to generate plots

- When you generate a figure that you might want to later include in a paper or presentation, save the raw data you used to generate it.
- When (inevitably) you later decide to change the formatting, you will be able to do so easily.

Here's how I make it easy to regenerate figures

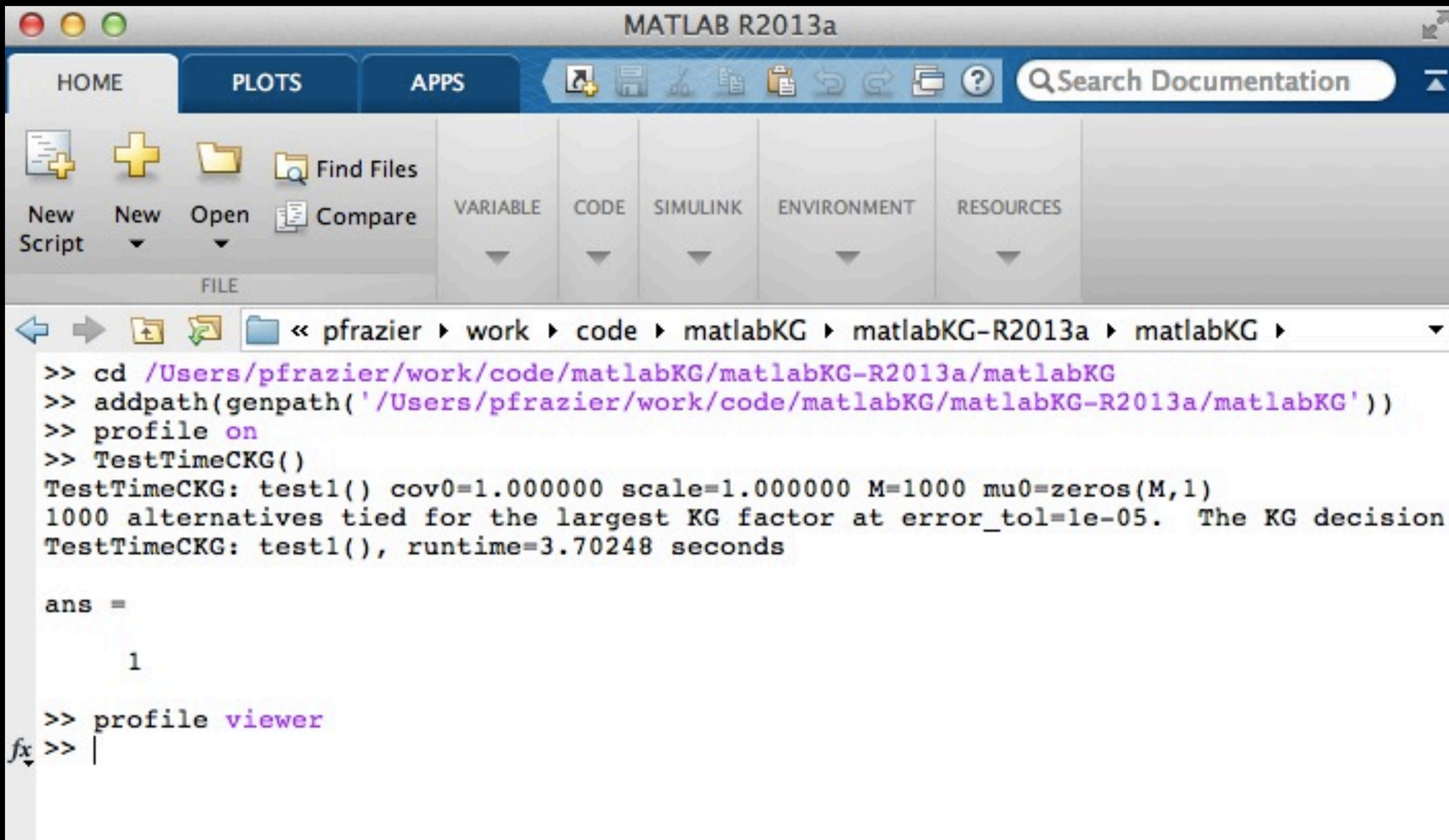
- When I get a figure I might want to re-use, I write a script that reads in the raw data, creates the figure, and writes the figure to a file.
- I save the data, script, and figure in the repository.
- Later it is easy to find the data & script to regenerate the figure because:
  - The script has the figure filename in it, and so will turn up in searches.
  - I usually name the script with a similar name to the figure.

```
% Creates the following plots, all in the fig/ directory:
%       plotOptimalVsEndTime_improvement_rest.pdf
%       plotOptimalVsEndTime_improvement_tissue.pdf
%       plotOptimalVsEndTime_prob_rest.pdf
%       plotOptimalVsEndTime_prob_tissue.pdf
% The "improvement" plots show the percentage improvement of the optimal over
% status quo, as a function of the time horizon, and letting the optimal
% schedule use the same number of checkups as the status quo over that
% time-span.
%
% The "prob" plots show the the probability of an emergency, again as a
% function of the time horizon, letting the optimal schedule use the same
% number of checkups as the status quo.
%
% There is one "improvement" plot and one "prob" plot for each patient sub-group
% (rest pain and tissue loss).
%
function plotOptimalVsEndTime()

% Values in paper for rest pain
% lambda = 0.0338;
% alpha = 0.450;

% New values for rest pain
lambda_rest = 0.0336;
alpha_rest = 0.242;

% Values in paper for tissue loss
% lambda_tissue = 0.0530;
% alpha_tissue = 0.9022;

% New values for tissue loss
lambda_tissue = 0.0389;
alpha_tissue = 0.6798;

make_plot(lambda_tissue,alpha_tissue,'tissue', 'Tissue Loss');
make_plot(lambda_rest,alpha_rest,'rest','Rest Pain');
end

function make_plot(lambda,alpha,file_extension,plot_title)
endTime = [2:52];
for i=1:length(endTime)
  foptimal(i) = CalculateOptimal(endTime(i),lambda,alpha);
  foriginal(i) = CalculateOriginal(endTime(i),lambda,alpha);
  improvement(i) = (foriginal(i) - foptimal(i)) / foriginal(i);
end

plot(endTime,foriginal,'r--', endTime,foptimal,'k-')
set(gca,'fontsize',18);
legend('optimal', 'status quo','Location','SouthEast');
title(plot_title);
xlabel('Time Horizon (4wk periods)');
ylabel('Prob(undetected failure)');
axis([1 52 0 0.8])
print('-dpdf', sprintf('fig/%s_%s.pdf', 'plotOptimalVsEndTime_prob', file_extension)
);

plot(endTime,improvement*100)
set(gca,'fontsize',18);
title(plot_title);
xlabel('Time Horizon (4wk periods)');
% ylabel('Percent improvement (status quo - optimal) / (status quo)');
ylabel('Percent improvement');
axis([1 52 0 max(improvement)*100*1.1])
```

To make your program faster, first understand why it is slow

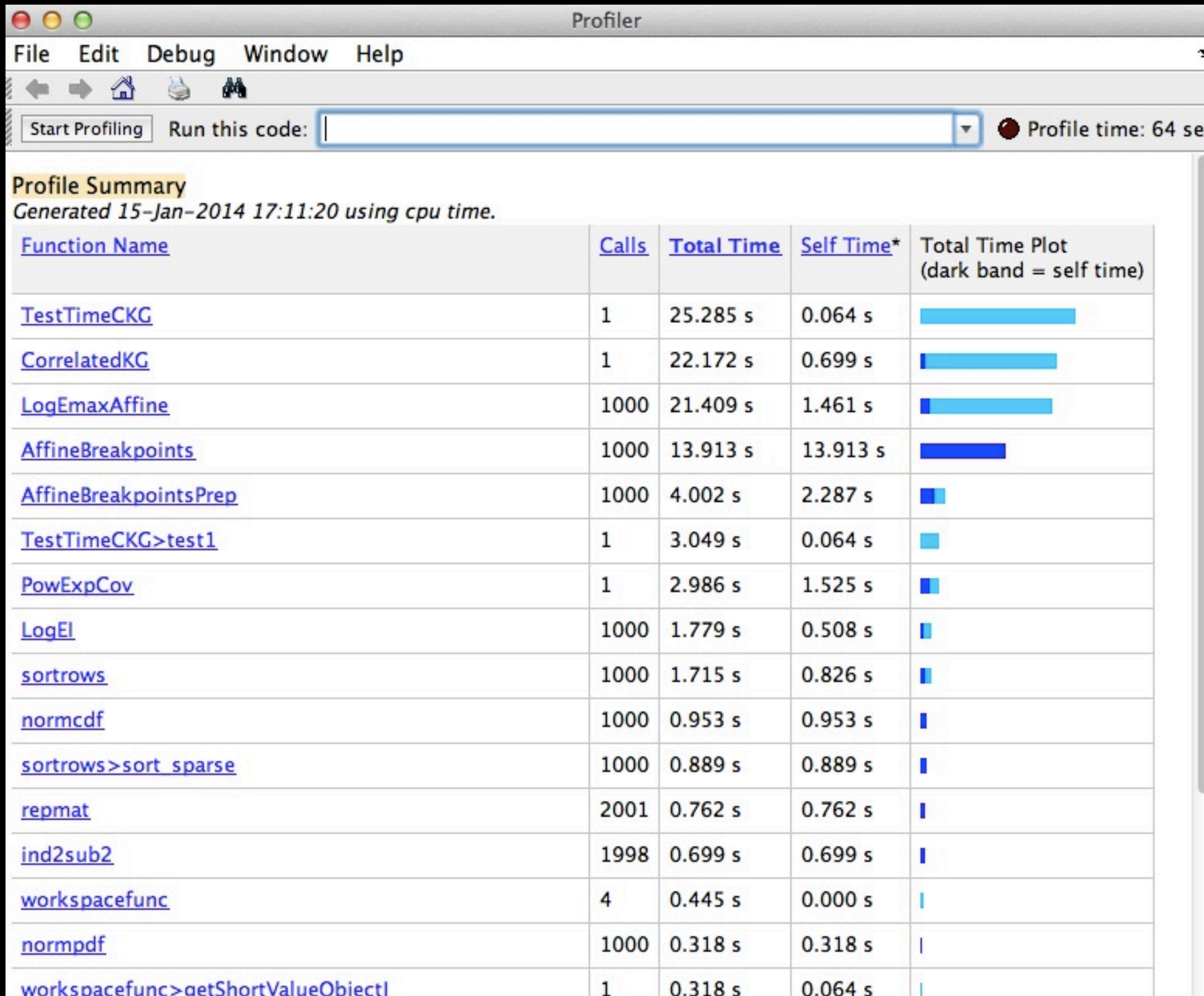
- Use **profiling** to observe which parts of your program contribute most to runtime.
- Understand a bit about **computer architecture**.
- Use **system monitoring tools** to observe CPU, disk and network to find which is the bottleneck.

Here's how to profile in Matlab



- Run "profile on" to start profiling.
- Run "profile off" to stop.
- Run "profile viewer" to see the results.

Here's how to profile in Matlab

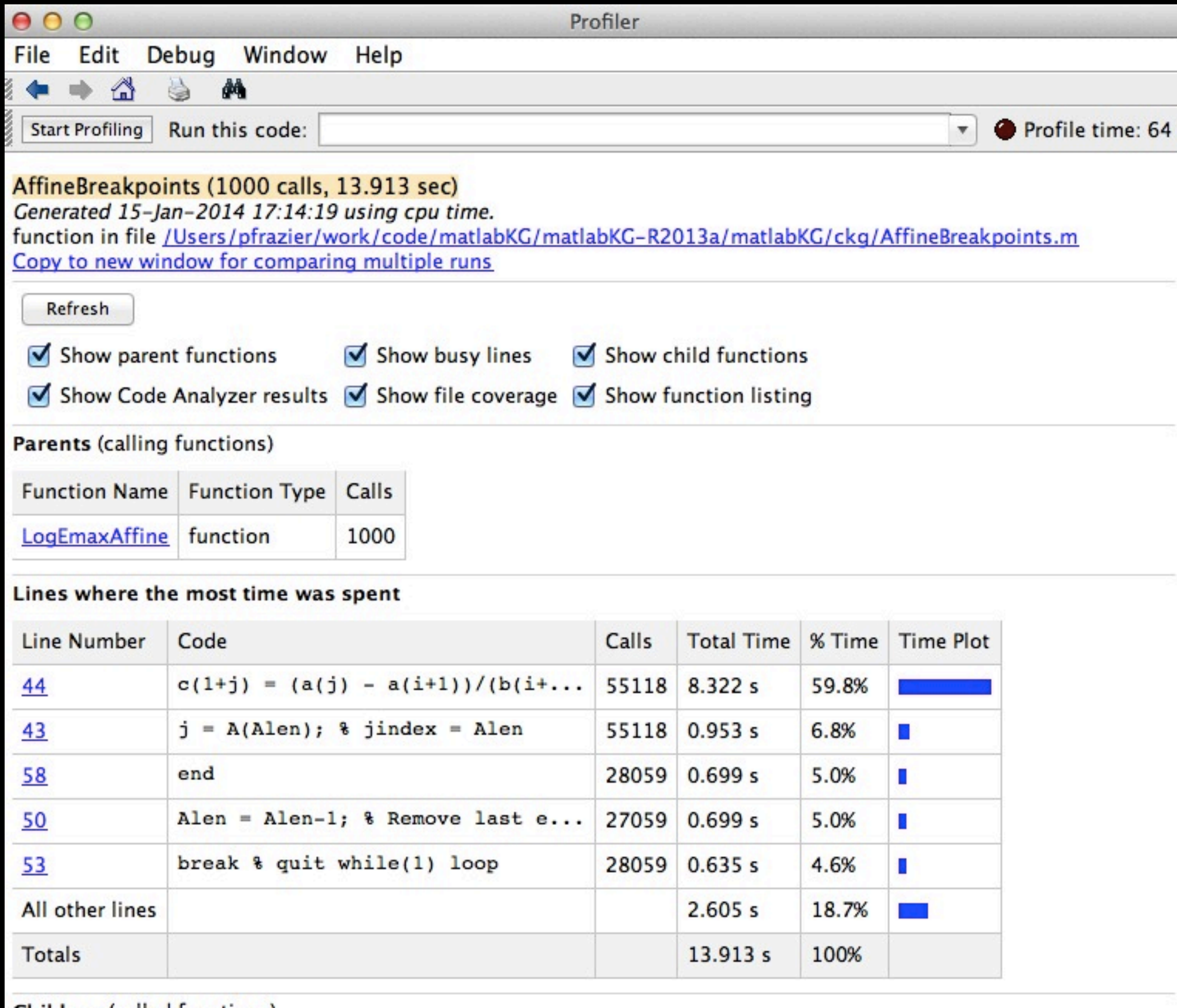


Profile Summary

Generated 15-Jan-2014 17:11:20 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
TestTimeCKG	1	25.285 s	0.064 s	
CorrelatedKG	1	22.172 s	0.699 s	
LogEmaxAffine	1000	21.409 s	1.461 s	
AffineBreakpoints	1000	13.913 s	13.913 s	
AffineBreakpointsPrep	1000	4.002 s	2.287 s	
TestTimeCKG>test1	1	3.049 s	0.064 s	
PowExpCov	1	2.986 s	1.525 s	
LogEI	1000	1.779 s	0.508 s	
sortrows	1000	1.715 s	0.826 s	
normcdf	1000	0.953 s	0.953 s	
sortrows>sort_sparse	1000	0.889 s	0.889 s	
repmat	2001	0.762 s	0.762 s	
ind2sub2	1998	0.699 s	0.699 s	
workspacefunc	4	0.445 s	0.000 s	
normpdf	1000	0.318 s	0.318 s	
workspacefunc>getShortValueObjectJ	1	0.318 s	0.064 s	

Here's how to profile in Matlab


Profiler

File Edit Debug Window Help

Start Profiling Run this code: Profile time: 64 s

AffineBreakpoints (1000 calls, 13.913 sec)

Generated 15-Jan-2014 17:14:19 using cpu time.

function in file /Users/pfrazier/work/code/matlabKG/matlabKG-R2013a/matlabKG/ckg/AffineBreakpoints.m

Copy to new window for comparing multiple runs

Refresh

☑ Show parent functions ☑ Show busy lines ☑ Show child functions

☑ Show Code Analyzer results ☑ Show file coverage ☑ Show function listing

Parents (calling functions)

Function Name	Function Type	Calls
LogEmaxAffine	function	1000

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
44	`c(1+j) = (a(j) - a(i+1))/(b(i+...`	55118	8.322 s	59.8%	
43	`j = A(Alen); % jindex = Alen`	55118	0.953 s	6.8%	
58	`end`	28059	0.699 s	5.0%	
50	`Alen = Alen-1; % Remove last e...`	27059	0.699 s	5.0%	
53	`break % quit while(1) loop`	28059	0.635 s	4.6%	
All other lines			2.605 s	18.7%	
Totals			13.913 s	100%	

If your Matlab program is too slow...

- Profile it first to find out what parts are causing speed issues.
- Replace loops by operations on matrices and vectors.
- Try the matlab compiler.
- Can you use more efficient algorithms?
- Rewrite the slow parts of your code, or all of your code, in C.

Here's how to profile in C and C++

- use the -pg and -g flags when compiling your source code into object files, and when linking your object files into an executable.
- When you run your program, it will create a file in the current directory called "gmon.out"
- Then run "gprof" with your executable's name as the first argument.

Here's an example in C

First, here's my test program

```
[pf98@asimov]~/profiling% cat myprog.c
#include <stdio.h>
#include <math.h>

double my_function1(double x) { return x+2; }
double my_function2(double x) { return log(x); }

int main() {
  int i;
  double x=0;
  for(i=0;i<100000000;i++) {
    x=my_function1(x);
    x=my_function2(x);
    if (i<20)
      printf("%f\n", x);
  }
  printf("%f\n", x);
}
```

Here’s an example in C

compiling with profiling on using -g and -pg (and also with the math library -lm)

running my program

my program computes some numbers

run gprof to get profiling output

here’s the profiling output

```
[pf98@asimov]~/profiling% gcc -o myprog myprog.c -g -pg -lm
[pf98@asimov]~/profiling% ./myprog
0.693147
0.990710
1.095511
1.129953
1.141018
1.144547
1.145670
1.146027
1.146140
1.146176
1.146188
1.146192
1.146193
1.146193
1.146193
1.146193
1.146193
1.146193
1.146193
1.146193
1.146193
[pf98@asimov]~/profiling% gprof myprog
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 44.74      0.31     0.31 100000000     3.13     3.13  my_function2
 28.87      0.52     0.20                             main
 23.09      0.68     0.16 100000000     1.62     1.62  my_function1
  4.33      0.71     0.03                             frame_dummy
```

If your C or C++ program is too slow...

- understand whether your code is CPU or I/O bound
- improve efficiency and use better algorithms in portions of the code identified as bottlenecks by profiling
- don’t forget to turn off profiling once you no longer need it --- it slows down the code
- compile with the -O3 flag to tell the compiler to aggressively optimize for speed.
- rewrite your code to run in parallel on more cores, or on a GPU
- understand CPU cache, and design your code to reduce cache misses

Understand a little bit about computer architecture

- Your computer has different ways of storing data, each with its own speed & size
  - Level 1 cache on the CPU: ~64KB, 0.5ns
  - Level 2 and higher caches, on and near the CPU: ~2MB, 5ns
  - RAM: ~4GB, 100ns
  - SSD Hard Disk: ~100GB, 150,000ns=150us
  - Traditional Hard Disk: ~1TB, 10,000,000ns=10ms

The time to access data affects your program's speed

- If you write your program to get more of its data from the CPU cache, and less from RAM, it will be faster.
- If you write your program to get more of its data from the RAM, and less from disk, it will be faster.

The time to access data affects your program's speed

- The operating system automatically figures out when & where a particular memory address will be stored (except for SSD vs. traditional hard disks).
- If you are only accessing a few KB of data at a time, this will mostly be served from CPU cache and will be fast.
- If you are jumping around within many MB of data, this will be slower as it will need to be served from RAM.
- If you are jumping around within many GB of data, this will need to be served from disk, even if you don't explicitly store it on disk in your code.

Use performance monitoring tools to check which resources are being used

- On UNIX and Mac OSX, I use top and sar. vmstat is also useful.
- On Windows, I use performance monitor.

Learn how to use UNIX & the cluster

- Using the full power of parallel computing accelerates computational studies.
- Being a command-line wiz saves time, but takes time to learn.
- At Cornell there is a short course in UNIX scripting that students can take.
- Perhaps there is something similar here.

Don’t be afraid to rewrite code

- Write code in a high-level language first, then rewrite in a low-level language later if needed for speed.
- Your code will eventually grow unwieldy. Depending on circumstances, it may be worth taking the time to refactor and redesign.

Further Reading

- Wilson et al. 2013, "Best Practices for Scientific Computing", http://arxiv.org/pdf/1210.0530v4.pdf