Lecture 23

Code Profiling and Debugging

(Some material adapted from Chris Simmons)

Tuesday, November 24th 2020

Performance Analysis

- Want a solution quickly
  - Some simulations are making predictions about time-sensitive things (hurricanes, disaster relief, etc).
- Want to resolve currently intractable scientific problems
  - Scientific discovery
- Want efficient code
  - This has a monetary impact
- Predictive science
  - Machine learning, uncertainty quantification, optimization, …
  - These require running many code executions

Why Profile Code?

- You want your program to run *faster*
- Usually, a program has one or more bottlenecks leading to slow execution time
- You could diagnose these manually by inserting timers into different parts of your code
    - This would be a nightmare!
- It would be awfully nice if there were a way to automatically diagnose how fast / slow different parts of the code are running
    - This is where code profilers come in

What are the Options?

- For common compiled languages (`C`, `C++`, `Fortran`), have a look at [`gprof`]
- In Python, you have [`cProfile` and `profile`]
- `cProfile` is a good choice as a default option with low overhead

Demo 1

```python
import cProfile

def convert(sentence):
    return sentence.split()

cProfile.run('convert("A very fine sentence.")')
```

Demo 1 Output

```
     5 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

Demo 1 Output

```
5 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

There were five total function calls.
It took `0.000` seconds (but not really).

Demo 1 Output

```
   5 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

Output is sorted by text string in the last column

Demo 1 Output

```
         5 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

`ncalls`: The number of times that function was called.

Demo 1 Output

```
   5 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

`tottime:` The total time spent in that function.
This does not include time in subfunctions.

Demo 1 Output

```
   5 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

`percall`: Time per call (`tottime / ncalls`).

Demo 1 Output

```
   5 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

`cumtime`: Cumulative time spent in this function and all of its subfunctions.

Demo 1 Output

```
   5 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

`percall: cumtime / primitive calls.`

A `primitive call` is a function call that was not induced by recursion.
In our example, everything is a `primitive call`.

Demo 1 Output

```
     5 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <stdin>:1(convert)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
```

`filename:lineno(function)`: The data of the function.

Writing and reading performance stats

```python
import pstats

cProfile.run('convert("A very fine sentence.")', 'pstats')

p = pstats.Stats('pstats')

p.print_stats()
```

Demo 2

```python
import numpy as np
import cProfile

def f(x):
    return x - np.exp(-2.0 * np.sin(4.0*x) * np.sin(4.0*x))

def dfdx(x):
    return 1.0 + 16.0 * np.exp(-2.0 * np.sin(4.0*x) * np.sin(4.0*x)) * np.sin(4.0*x) * np.cos(4.0*x)

def dfdx_h(x, epsilon):
    return (f(x + epsilon) - f(x)) / epsilon


def main():
    # Start Newton algorithm
    xk = 0.0 # Initial guess
    tol = 1.0e-08 # Some tolerance
    max_it = 100 # Just stop if a root isn't found after 100 iterations
    h = 1.0e-09

    root = None # Initialize root
    for k in range(max_it):
        delta_xk = -f(xk) / dfdx(xk) # Update Delta x_{k}
        if (abs(delta_xk) <= tol): # Stop iteration if solution found
            root = xk + delta_xk
            print("Found root at x = {0:17.16f} after {1} iterations.".format(root, k+1))
            break
        print("At iteration {0}, Delta x = {1:17.16f}".format(k+1, delta_xk))
        xk += delta_xk # Update xk

cProfile.run('main()')
```

Breakout Room (10 minutes)

- Change the script to use the finite difference derivative in the Newton solver
- Run the profiler
- Did anything change?
  - Results?
  - Timing?
  - Function calls?

Debugging

- There is no bug-free code
  - You will introduce bugs
  - Other members of the community will introduce bugs
  - Bugs even live in commercial codes
- Bugs can:
  - Prevent a code from running at all
  - Prevent a code from running well
  - Lead to incorrect results and predictions
- Good debugging skills will make you a more efficient and confident programmer

Defensive Programming

- Check function return codes for errors (more useful in C, C++, Fortran)
- Check input values
  - We discussed this in our testing unit
  - Check “impossible” values
- Write out the control parameters to a file
  - This helps you keep track of your runs
  - Also helps for experimental repeatability
- Check for “non-physical” results
  - This applies beyond the physical sciences
  - For example, for negative chemical concentrations
  - Or check for negative number of people
- Employ the techniques we’ve discussed throughout the semester
  - Write regression tests, use version control, write modular code, document, error checking

Some common indicators of bugs

- Build errors (for compiled code) 😒
- Improper memory reads / writes (again, usually met in C / C++) 😫
- Illegal operations (division by zero, etc) 😞
- Infinite loops 😬
- I/O errors 😳
- Algorithmic errors 🙀
- Poor performance 😭

The Debugging Process

- Start with defensive programming
    - You will still get bugs, just not nearly as many
- Basic steps:
    - Determine that there is a bug somewhere
    - Isolate the source of the bug (a.k.a. find the bug)
    - Identify the cause of the bug (how did this happen?)
    - Determine a fix for the bug (how do we remove this bug?)
    - Fix the bug and test
- These steps are not always so easy
- A debugger can help

Debuggers

- Command line debuggers are tools to aid in diagnosing problems
- The GNU Debugger Project
  - Very powerful
  - Works with many languages (not Python)
  - “allows you to see what is going on `inside' another program while it executes”
- pdb - The Python Debugger
- These debuggers are a front-end for the application
  - Step though the code and examine: variables, arrays, functions, etc
- Have the opportunity to investigate the run-time behavior of the application

Debugging: Important Commands and Concepts

- Show program backtraces
  - The calling history up to the current point
- Set breakpoints
- Display values of individual names
- Set new values
- Step through the program

Breakpoints

- A breakpoint is a pseudo-instruction that you can insert at any place during a debugging session
- The debugger will interpret the breakpoint
- The program execution hits a breakpoint, the debugger will pause the program so you can:
    - Inspect names
    - Set and / or clear breakpoints
    - Continue execution
- There are also conditional breakpoints
    - Program pauses only if the breakpoint’s condition holds
    - e.g. an expression is true, the breakpoint has been crossed “N” times, an expression changed its value

`pdb` demo

```
>>> import pdb # import pdb
>>> import script # import our module
>>> pdb.run('script.main()') # attach debugger
```

Getting Familiar

```
>>> pdb.run('script.main()')
> <string>(1)<module>()
(Pdb) continue
```

- Attach debugger to main script
- Debugger enters module and pauses by default
- We type continue to just run through everything

Stepping Through

```
(Pdb) s
--Call--
> script.py(14)main()
-> def main():
(Pdb) s
> script.py(16)main()
-> xk = 0.0 # Initial guess
```

- Take a step to the next line
- We’re at the main function
- Take a step into the main function
- We’re at the initial guess

Exploring

```
-> xk = 0.0 # Initial guess
(Pdb) s
> script.py(17)main()
-> tol = 1.0e-08 # Some tolerance
(Pdb) p xk
0.0
```

- Execute line setting initial guess
- Go to next line
- Before executing, print out value of `xk` just to check

Getting our bearings

```
-> for k in range(max_it):
(Pdb) l
 17             tol = 1.0e-08 # Some tolerance
 18             max_it = 100 # Just stop if a root isn't found after 100 iterations
 19             h = 1.0e-09
 20
 21             root = None # Initialize root
 22  ->           for k in range(max_it):
 23                 delta_xk = -f(xk) / dfdx(xk) # Update Delta x_{k}
 24                 if (abs(delta_xk) <= tol): # Stop iteration if solution found
 25                     root = xk + delta_xk
 26                     print("Found root at x = {0:17.16f} after {1} iterations.".format(root, k+1))
 27                     break
```

- We stepped through a bunch of lines of code
- Let’s list the lines around the current line
- We can see where we are and we can see a few lines above and below

Setting a breakpoint

- We're worried that our function fk isn't doing the right things
- Let's set a breakpoint at line 24
- Then we'll continue to that point
- Once we get to the breakpoint we can examine things more

```
(Pdb) b 24
Breakpoint 2 at script.py:24
(Pdb) continue
> script.py(24)main()
-> if (abs(delta_xk) <= tol): # Stop iteration if ...
```

More exploration

```
(Pdb) p delta_xk
1.0
(Pdb) p f(xk)
-1.0
(Pdb) p dfdx(xk)
1.0
```

- Have a look at the step value
- Check the output of `f()`
- Check the output of `dfdx()`
- Things look fine so far!
- Let’s put in another breakpoint

Going into the functions

```
(Pdb) b 23
Breakpoint 3 at script.py:23
(Pdb) continue
At iteration 1, Delta x = 1.0000000000000000
> script.py(23)main()
-> delta_xk = -f(xk) / dfdx(xk) # Update Delta x_{k}
```

Inspecting the function

```
(Pdb) s
--Call--
> script.py(4)f()
-> def f(x):
(Pdb) s
> script.py(5)f()
-> return x - np.exp(-2.0 * np.sin(4.0*x) * np.sin(4.0*x))
(Pdb) s
--Return--
> script.py(5)f()->0.6819351651504464
-> return x - np.exp(-2.0 * np.sin(4.0*x) * np.sin(4.0*x))
```

Other fun things: Exploring names

```
(Pdb) whatis f
Function f
(Pdb) whatis xk
<class 'numpy.float64'>
```

Other fun things: listing breakpoints

```
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at script.py:24
    breakpoint already hit 4 times
2   breakpoint   keep yes   at script.py:24
3   breakpoint   keep yes   at script.py:23
    breakpoint already hit 2 times
```

Other fun things: disabling breakpoints

```
(Pdb) disable 1
Disabled breakpoint 1 at script.py:24
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep no    at script.py:24
    breakpoint already hit 4 times
2   breakpoint   keep yes   at script.py:24
3   breakpoint   keep yes   at script.py:23
    breakpoint already hit 2 times
```

Finishing up

```
(Pdb) continue
At iteration 3, Delta x = 0.0782795784669513
At iteration 4, Delta x = -0.0271256514726681
At iteration 5, Delta x = -0.0012447280598220
At iteration 6, Delta x = -0.0000049668585411
Found root at x = 0.8560316824308374 after 7 iterations.
```

- In this example, we disable our breakpoints and just ran everything to the end
  - Easy because this is a little code
- Alternatively, when we were done, we could have just quit (`q`)

```
(Pdb) q
>>>
```

Summary / Debrief

- The debugger can make debugging more fun!
- We only covered the most basic tasks today
- There are many other things you can do
- Here are some resources
    - `pdb` - The Python Debugger
    - `pdb` cheatsheet
    - Python Call Graph - Pretty nifty way to visualize what your code is doing