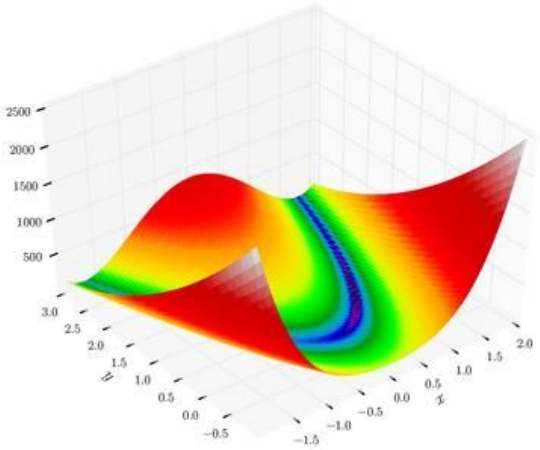
CSE 374: Lecture 27

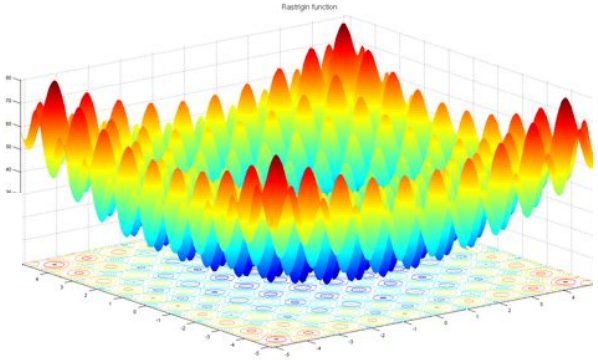
Profiling and memory

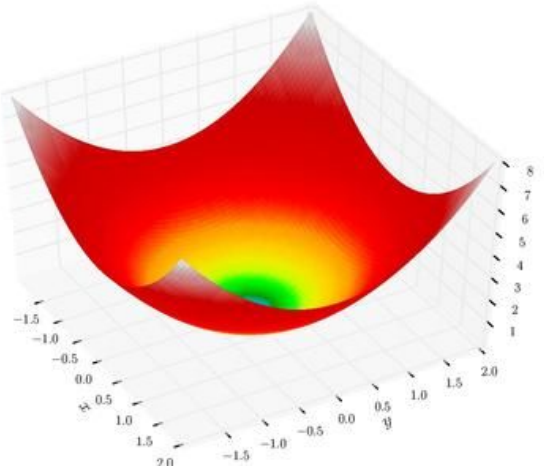
Particle Swarm Optimization

- Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling.
- Used to find the global optimum of potentially non-convex functions.
  - Optimize control settings
  - Fit data to functions
  - Find low energy solutions
    - Low energy often matches the natural solution
- Function optimization is usually an iterative algorithm
  - Coding inefficiencies add up.

Finding minima







Not to be confused with code or algorithm optimization

Minimize memory usage, computation time, or both

- Examination of behavior of a running program
- Tally of memory allocation
- Record of run time, including breakdown of where the time is spent.

Can use a variety of techniques (hardware interrupts, code tooling, performance counters)
Trace: stream of recorded events, proportional to execution time
Profile: statistical summary of event, proportional to code size

Basics

1. Write code
2. Run test cases (benchmarks)
3. Python clint.py
4. Valgrind

Benchmarking v. Profiling

Benchmarking collects statistics on specific sample problems

(Ex. objective functions are standard benchmark functions for optimization)

- ➔ Number of iterations until convergence
- ➔ Likelihood of finding solution
- ➔ Run time
- ➔ Memory usage

Benchmarking can be very useful for measuring performance on subsequent deliveries

Profiling Tools

- Investigate run-time behavior of code at different points
- Checks time taken by instructions from machine language to high-level functions
  - actual time
  - number of calls to the instruction
- Flat profiler - computes average call times, does not break down calls
- Call graph profiler - shows chains based on called functions

Insertion v. Sampling profilers

Insertion:

- Place specific profiling code in program
- Can be used on various platforms
- Accurate
- Requires recompilation and relinking
- Will affect performance

Sampling:

- Monitoring or snap-shotting at specific intervals
- No modification of code
- Less accurate - limited by sampling rate
- Very small methods often missed
- Not great for memory

$gprof

Gnu profiling tool

Compile with `$gcc -pg` flag

```
$./mainopt
```

Creates gmon.out

Run profiler with

```
$gprof ./mainopt
```

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
64.87	0.22	0.22	2040000	0.00	0.00	update_vel
14.74	0.27	0.05	2008462	0.00	0.00	rastrigin
11.79	0.31	0.04	2040000	0.00	0.00	update_pos
5.90	0.33	0.02	102	0.20	3.34	optimize
2.95	0.34	0.01	51000	0.00	0.00	update_gb
0.00	0.34	0.00	20132	0.00	0.00	rosenbrock
0.00	0.34	0.00	20131	0.00	0.00	spherefunc
0.00	0.34	0.00	102	0.00	0.00	initialize_opt

$valgrind --tool=callgrind

```
$valgrind --tool=callgrind ./mainopt
```

Creates callgrind.out.X
You can read output file

But its tricky; try:

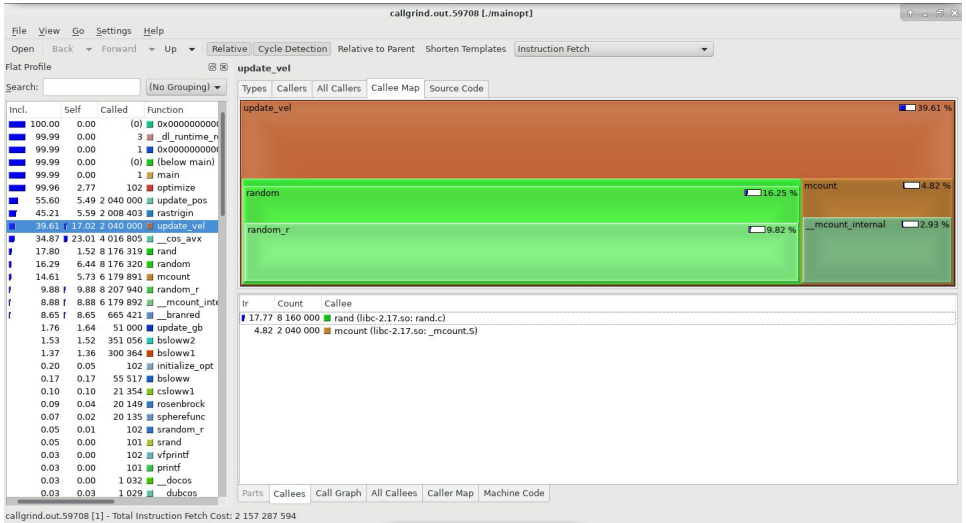
```
$kcachegrind callgrind.out.X
```

(Must install cachegrind:

```
$sudo yum install kcachegrind
```

Or, on Ubuntu:

```
$sudo apt-get install kcacehgrind
```



Observe

- Which methods are being called the most
  - these may not necessarily be the "slowest" methods!
- Which methods are taking the most time relative to the others
  - common problems
    - inefficient unbuffered I/O
    - poor choice of data structure
    - recursion call overhead
    - unnecessary re-computation of expensive information, or unnecessary multiple I/O of same data

Course Topics

- ★ Linux, Shells & Emacs, Redirection
- ★ Shell variables and scripting
- ★ Regular expressions, grep & sed
- ★ C (parameters, scope, malloc, structs, lists, trees)
- ★ Stack & Heap structures, memory management
- ★ Debugging (gdb), Valgrind, Testing
- ★ Makefiles
- ★ Git
- ★ C++ (object oriented programming, classes, namespaces)
- ★ Threads and concurrency
- ★ Profiling and process review