Process System Engineering #5

#03150796 Amane Suzuki

November 4, 2015

1 Abstract

We use the Steepest Descent Method to find the minimum of functions. Steepest Descent Method can calculate the minimum of simple funcion enough fast and accurate. However when find the minimum of little complicated function, processing time is much longer depending on initial condition.

2 Algorithm

We use Steepest Descent Method to find the minimum of following two functions.

$$f(x, y) = (x - 1)^2 + 50(y - 1)^2 \tag{1}$$
$$g(x, y) = (x - 1)^2 + 100(x^3 - y)^2 \tag{2}$$

Figure 1 shows a flow of Steepest Descent Method.

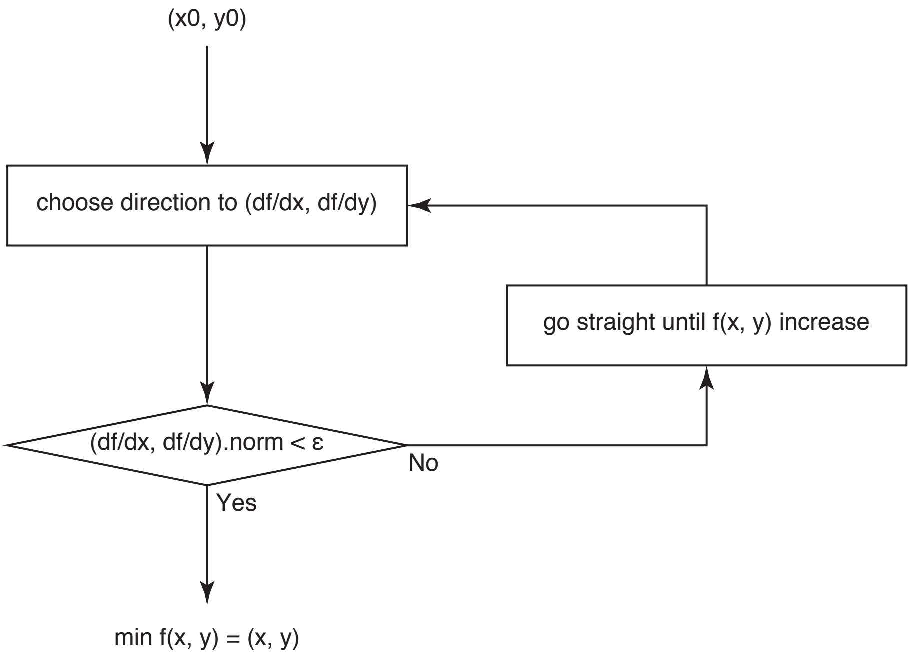

Figure 1: The flow chart of steepest decent method.

First, this method choose direction where function decreases steepest.

$$(\text{directionvector}) = -\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \tag{3}$$

If the direction vector is sufficiantly small, this method regard (x, y) of that time as the minimum of the funcion. After decided direction vector, it try going straight until the function increases. Repeating these operations, (x, y) approach to the minimum.

Steepest Descent Method is easy to coding because it only uses first derivation. It has, however, a few demerits.

- Depending on the initial (x, y), it approach to *local* minimum.
- Steepest Descent does NOT mean fastest method to find minimum.
- It needs partial differential of the funcion.

3 Result

3.1 Find minimum of $f(x, y)$ with $(x_0, y_0) = (0, 0)$

```
$ ruby optimize.rb
Vector[0.9999999999998566, 1.0000000000000009]
0.015027 sec
```

Numerical solution is $(x, y) = (0.9999999999998566, 1.0000000000000009)$. It means this method can find the correct minimum of $f(x, y)$. (cf. Analytical solution is $(x, y) = (1, 1)$)

3.2 Benchmark of finding $min(f(x, y))$

Table 1 shows the result of benchmark. In this benchmark, we choose 10000 initial condition at random, test whether it approach the true minimum, and measure the processing time.

Pass means the result is correct. Fail means the result is wrong. Timeout means processing time is over 3 second. Average Time is the agerage time of *Pass* cases.

Pass	10000
Fail	0
Timeout	0
Average Time	0.023 sec

Table 1: Benchmark of finding minimum of $f(x, y)$

The result shows that this method is useful finding the minimum of function f.

3.3 Find minimum of $g(x, y)$ with $(x_0, y_0) = (0, 0)$

```
$ ruby optimize.rb
Vector[0.9984717014948186, 0.9954053544707571]
1.33843 sec
```

Numerical solution is $(x, y) = (0.9984717014948186, 0.9954053544707571)$. It means this method can find the correct minimum of $g(x, y)$. (cf. Analytical solution is $(x, y) = (1, 1)$)

Pass	7663
Fail	0
Timeout	2337
Average Time	1.04 sec

Table 2: Benchmark of finding minimum of $g(x, y)$

3.4 Benchmark of finding $min(g(x, y))$

Table 2 shows the result of benchmark.

When finding the minimum of function f, this method has enough speed and accuracy. In contrast, when finding the minimum of function g, there is 2337 timeouts.

Figure 2 shows the plot of initial conditions that goes timeout.

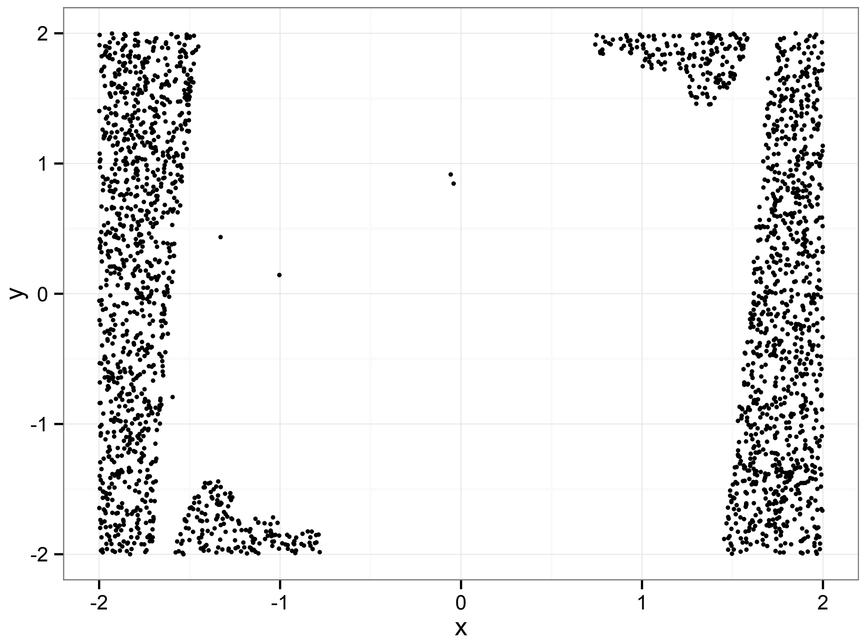

Figure 2: The initial conditions of Timeout cases.

4 Discussion

Steepest Descent Method can calculate the minimum of simple funcion enough fast and accurate. Using to a function which has deep valley (e.g. $g(x, y)$), we have to choose initial condition carefully or execute a lot of trials.

Proper initial condition doesn't mean coordinate which is near solution and isn't easy to find, therefore we should calculate in a variety of initial conditions.

5 Source Program

Listing 1: optimize.rb

```ruby
require "matrix"
require "benchmark"
require "timeout"
require "csv"

DEBUG = false

ANALYTICAL_SOLUTION = Vector[1.0, 1.0]
STEP = 0.0005
DELTA = 0.00025
EPSILON = 0.001
TIMEOUT = 3.0
TOLERANCE = 0.01

TIMEOUT_CSV_PATH = "./timeouts.csv"

f = lambda{ |coordinate|
  x=coordinate[0]
  y=coordinate[1]
  (x-1)**2+50*(y-1)**2
}

g = lambda{ |coordinate|
  x=coordinate[0]
  y=coordinate[1]
  (x-1)**2+100*(x**3-y)**2
}

# use steepest descent method
def optimize(func, coordinate)
  begin
    # grad f
    vector = Vector[
      (func.call(coordinate+Vector[DELTA, 0.0])-func.call(coordinate
          -Vector[DELTA, 0.0]))/DELTA/2,
      (func.call(coordinate+Vector[0.0, DELTA])-func.call(coordinate
          -Vector[0.0, DELTA]))/DELTA/2
    ]

    coordinate = line_search(func, coordinate, -STEP*vector)
  end while vector.norm > EPSILON
  puts coordinate if DEBUG
  return coordinate
end

# go straight until value increase
def line_search(func, coordinate, vector)
  while func.call(coordinate+vector) < func.call(coordinate)
    coordinate += vector
  end
```

```
  return coordinate
end

def benchmark(func, n)
  pass_num = 0
  fail_num = 0
  timeout_num = 0
  total_time = 0
  timeouts = []
  # optimize test 100 times
  for i in 1..n
    print "#{i}: "

    # start from random coordinate
    x = Random.rand(-2.0..2.0).round(3)
    y = Random.rand(-2.0..2.0).round(3)
    start = Vector[x, y]
    result = Vector[0, 0]

    begin
      timeout(TIMEOUT) {
        time = Benchmark.realtime do
          result = optimize(func, start)
        end

        if (result-ANALYTICAL_SOLUTION).norm < TOLERANCE
          pass_num += 1
          total_time += time
          print "pass in #{time.round(3)}s"
        else
          fail_num += 1
          print "fail => result: #{result}"
        end
      }
    rescue Timeout::Error
      print "timeout => start: #{start}"
      timeout_num += 1
      timeouts << start
    end

    print "\n"
  end
  puts "pass: #{pass_num}"
  puts "fail: #{fail_num}"
  puts "timeout: #{timeout_num}"
  puts "average time: #{(total_time/pass_num).round(3)}s"

  CSV.open(TIMEOUT_CSV_PATH, "wb") do |csv|
    timeouts.each do |vector|
      csv << [vector[0], vector[1]]
    end
  end
  puts "output timeout vectors"
end
```

```
# start = Vector[0,0]
# time = Benchmark.realtime do
# result =optimize(g, start)
# end
# puts "#{time} sec"
benchmark(g, 10000)
```