Optimization:
Gradient Descent

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/~jburkardt/classes/math2070_2019/optimization_gradient_descent/optimization_gradient

This cautious skater has not chosen the optimal descent direction!

A Minimization Problem

Given a function $f(x)$ with derivative $f'(x)$, find a value x for which $f(x)$ attain its minimum value..

1 Minimize a quartic function using gradient descent

Consider the function named *quartic()* whose formula is:

$$f(x) = 2x^4 - 4x^2 + x + 20$$

Suppose $f(x)$ measures a cost and we seek a value x which minimizes it.

The fact that *quartic()* is a polynomial means that we could seek zeroes of the derivative. However, this would not be a solution for a **general** function $f(x)$; this polynomial is just a convenient example.

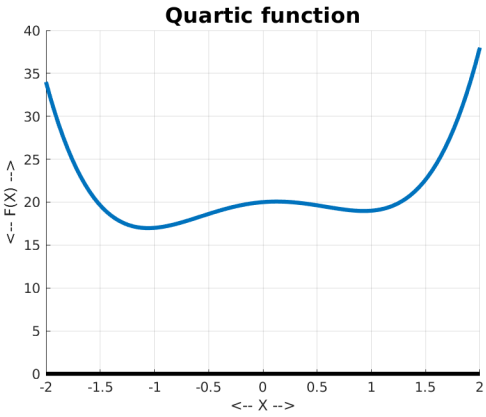
We might start at a point x in $[-2, +2]$ that we suspect is close to the minimizer. Our next step could be to move a little to the left or right from our starting point. If we happen to know $f'(x)$, then:

- if $0 < f'(x)$, then $f(x)$ decreases to the left: $\swarrow$;
- if $f'(x) < 0$, then $f(x)$ decreases to the right: $\searrow$;

For our quartic function, the derivative is

$$f'(x) = 8x^3 - 8x + 1$$

If we started our investigation at $x = 1.5$, then $f(1.5) = 22.625$, and $f'(1.5) = 16.0$ so for our next estimate of the minimizer, we should move a little to the left.



The quartic() function

This is the heart of the **gradient descent method**. From your current estimate, take a small step, related to the **size** of the derivative, opposite to the **direction** of the derivative. Keep doing that until the stepsize gets so small that you are evidently near a critical point. If your stepsize causes the function value to increase, back up and try a smaller stepsize.

We saw a similar idea of retrying the step when we looked at the damped Newton method.

2 Gradient descent pseudocode for 1D case

Pseudocode for a gradient descent method might look like this,

```
gradient_descent1 ( f, df, x, r, dxtol, dftol, itmax )

# gradient descent for a function of 1 parameter

  it = 0

  Loop1:

    if |df(x)| < dftol ) return (success)
    xold = x
    beta = 1.0

    Loop2:
      it = it + 1
      if ( itmax < it ) return (failure)
      dx = - beta * r * df(xold)
      if ( |dx| < dxtol ) return (success)
      x = xold + dx
      if f(x) < f(xold) break Loop2
      beta = beta / 2
      if ( beta < 1 / 1024 ) return (failure)
    Loop2 end

  Loop1 end
```

Listing 1: Pseudocode for gradient descent in 1D.

The input quantities `f` and `df` define the function and its derivative, `x` is the starting point, `r` is the *learning rate*, a sort of stepsize control, `dxtol` and `dftol` are tolerances for the minimum size of the step and derivative, and `itmax` limits the number of iterations.

3 Example: Applying gradient descent to quartic()

For our quartic function, we prepare files *quartic.m* and *quartic_df.m* defining $f(x)$ and $f'(x)$:

```
function value = quartic ( x )
  value = 2.0 * x.^4 - 4.0 * x.^2 + x + 20.0;
  return
end
```

Listing 2: quartic.m defines f(x).

```
function value = quartic_df ( x )
  value = 8.0 * x.^3 - 8.0 * x + 1.0;
  return
end
```

Listing 3: quartic_df.m defines f'(x)

We call `gradient_descent1()` with a starting guess `x0` in our chosen interval of $[-2, +2]$. The sizes of `r`, `dxtol` and `dftol` are just guesses for "small enough". We can adjust them after we run the code once:

```
  x0 = -1.6;
  r = 0.05;
  dxtol = 0.001;
  dftol = 0.001;
  itmax = 100;
  [ x, it ] = gradient_descent1 ( @(x)quartic(x), @(x)quartic_df(x), x0, r, dxtol, dftol,
      itmax );
```

Listing 4: Calling gradient_descent1 for the quartic function.

Our results are:

it	x	f(x)	f'(x)
0	-1.5	19.625	-14
1	-0.8	17.4592	3.304
2	-0.9652	17.044154	1.5280722
3	-1.0416036	16.972825	0.29222552
4	-1.0562149	16.970507	0.023297948
5	-1.0573798	16.970493	0.00139357

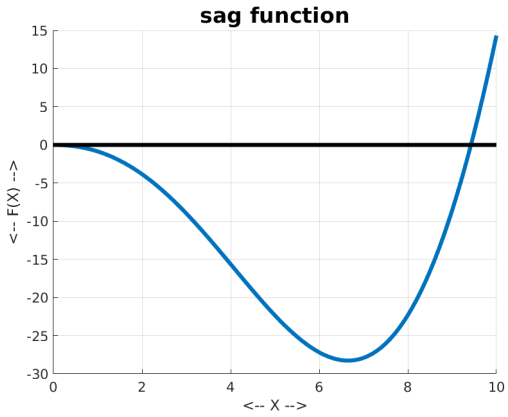
We end up on the left side of the plot, which seems likely since our starting point was nearest to that local minimum.

4 Exercise: Applying gradient descent to sag()

The example function *sag()* has the definition:

$$f(x) = x^2 \cos(\frac{x + 3\pi}{4})$$

A plot suggests that there is a minimizer between 0 and 10:



The sag() function

Create files *sag.m* and *sag_df.m* defining $f(x)$ and $f'(x)$, and call `gradient_descent1()` with a starting point in $[0, 10]$ and see if you can find a value of x for which $f(x)$ is minimized.

5 Gradient Descent in Higher Dimensions

Now let's consider what happens if we have a function of multiple variables. For simplicity, let's consider the two-dimensional case, which we may think of as involving a function $f(x_1, x_2)$ or $f(x, y)$. An example of such a function is named *hex2()*, and has the form

$$f(x, y) = 2x^2 - 1.05x^4 + x^6/6 + xy + y^2$$

We can ask for a minimizer of this function, which we suspect is somewhere in the range $-3 \leq x, y \leq +3$. As before, we will use derivative information, known as the *gradient*, symbolized by ∇f. For our problem, this information is $\left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$ with values:

$$\frac{\partial f}{\partial x} = 4x - 4.2x^3 + x^5 + y$$
$$\frac{\partial f}{\partial y} = x + 2y$$

6 Gradient descent pseudocode for multiple dimensions

The gradient descent code for multiple dimensions is almost the same, except for two places where we need to use a norm instead of an absolute value; also `x`, `xold`, `dx` and `df()` are vectors now, rather than scalars.

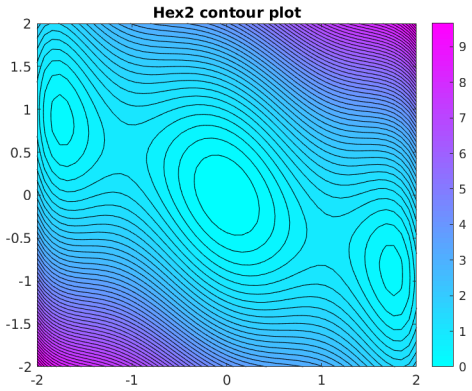
```
1  gradient_descent2 ( f(), df(), x, r, dxtol, dftol, itmax  )
2
3  # gradient descent for a function of multiple parameters
4
5    it = 0
6
7    Loop1:
8
9      if ||df(x)|| < dftol ) return (success)
10     xold = x
11     beta = 1.0
12
```

```
    Loop2:
      it = it + 1
      if ( itmax < it ) return (failure)
      dx = - beta * r * df(xold)
       if ( ||dx|| < dxtol ) return (success)
      x = xold + dx
      if f(x) < f(xold) break Loop2
      beta = beta / 2
      if ( beta < 1 / 1024 ) return (failure)
    Loop2 end

  Loop1 end

  Return x, it
```

Listing 5: Pseudocode for gradient descent in multiple dimensions.

7 Example: Applying gradient descent to hex2()

A contour plot suggests that there are three minimizers in the range $-2 \leq x, y \leq +2$.



Contour plot of the hex2() function

We prepare files *hex2.m* and *hex2_df.m*. The input `x` and output `df` are now vectors of length 2:

```
function value = hex2 ( xy )
  x = xy(1);
  y = xy(2);
  value = 2.0 * x.^2 - 1.05 * x.^4 + x.^6 / 6.0 + x .* y + y.^2;
  return
end
```

Listing 6: Definitions of f(x) for hex2().

```
function value = hex2_df ( xy )
  x = xy(1);
  y = xy(2);
  value = [ ...
    4.0 * x - 4.2 * x.^3 + x.^5 + y, ...
    x + 2.0 * xy ];
  return
end
```

Listing 7: Definitions of f'(x) for hex2().

We call `gradient_descent2()` with a starting point `x0` inside our interval, and tentative tolerances:

```
x0 = [ 2.0, 1.5 ];
r = 0.10;
dxtol = 0.00001;
dftol = 0.001;
itmax = 100;
[ x, it ] = gradient_descent2 ( @(x)hex2(x), @(x)hex2_df(x), x0, r, dxtol, dftol, itmax );
```

Listing 8: Calling gradient_descent2 for hex2().

Our results are:

it	x	y	f(x)	dfdx	dfdy
0	2.000000	1.500000	7.116666	7.970000	5.000000
1	1.210000	1.000000	3.410503	0.993186	3.210000
2	1.110681	0.679000	2.397414	1.057327	2.468681
3	1.004948	0.432131	1.741589	1.214253	1.869212
4	0.883523	0.245210	1.277457	1.420986	1.373944
..	...	...	...		...
41	0.000248	-0.000599	3.340249e-07	0.000393	-0.000950
42	0.000209	-0.000504	2.364863e-07	0.000331	-0.000800

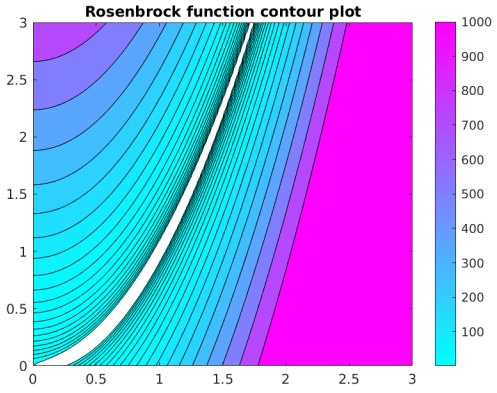
Here the results look pretty satisfactory.

8 Exercise: Applying gradient descent to rosenbrock()

The example function *rosenbrock()* has the definition:

$$f(x,y) = 100(y - x^2)^2 + (1 - x)^2$$

A contour plot suggests that there is a minimizer in the range $0.0 \leq x, y \leq 3.0$.



Contour plot of the Rosenbrock function

Create files *rosenbrock.m* and *rosenbrock_df.m* defining $f(x)$ and $f'(x)$; call `gradient_descent2()` starting at $[1.25, 1.50]$. Start with `r=0.0025` and `itmax=100`. You will find convergence very slow! Increase `itmax` by 100 several times, until you get an approximate minimizer (x, y) such that $|f(x, y)| < 0.00001$.

9 Gradient Descent for data fitting

Many machine learning problems start with a mass of data, in which one y is approximately determined by one or more variables that we think of as x. For instance, we may have n pairs of values (x_i, y_i). We look for parameters b and m in a linear model $y = b + m * x$ that approximates our data well.

The numbers b and m are unknown. For any particular i, we will measure the error e_i as the square of the difference between the model's prediction based on x_i, and the actual data y_i:

$$e_i(b, m) = (y_i - b - mx_i)^2$$

Now we want to minimize the sum of all those functions. We call this single function $E(b, m)$:

$$E(b, m) = \sum_{i=1}^{n} e_i(b, m)$$

We want to make this problem this look like a typical case for gradient descent. We start by defining a single variable $bm = [b, m]$, which packs all our variables into one. Correspondingly, we can now write $f(bm) = f([b, m]) = E(b, m)$. We need to evaluate $f(bm)$ and $df(bm)$:

```
function value = ford_f(bm)
  b = bm(1);
  m = bm(2);
  [ x, y ] = ford_data ( );
  f = sum ( ( y - b - m * x ).^2 );
  return
end
```

Listing 9: Evaluate f(bm) for Ford data

```
def value = ford_df(bm)
  b = bm(1);
  m = bm(2);
  [ x, y ] = ford_data ( );
  dfdb = - 2.0 * sum ( ( y - b - m * x ) );
  dfdm = - 2.0 * sum ( ( y - b - m * x ) .* x );
  value = [ dfdb, dfdm ];
  return
end
```

Listing 10: Evaluate gradient $\nabla(f)$ for Ford data

and now we need to write a function `ford_data()` which returns a copy of our data. The gradient descent method does not perform well if the variables have very different scales. To be safe, we can always rescale each column to lie between 0 and 1. However, this means that our values of `b` and `m` will apply to the normalized data. If we want to write the formula in terms of the original data, we have to undo the normalization (a cleanup task which we will not do here!)

```
function [ x, y ] = ford_data ( )

  data = np.loadtxt ( 'ford_data.txt' )

  x = data(:,1);
  y = data{:,2);

  x = ( x - min ( x ) ) / ( max ( x ) - min ( x ) );
  y = ( y - min ( y ) ) / ( max ( y ) - min ( y ) );

  return
end
```

Listing 11: Return the values of the Ford data

After writing these three procedures, we can call `gradient_descent2()` as before:

```
bm0 = [ 0.5, 0.0 ];
r = 0.01;
dxtol = 0.001;
dftol = 0.001;
itmax = 1000;
[ bm, it ] = gradient_descent2 ( @(x)ford_f(x), @(x)ford_df(x), bm0, r, dxtol, dftol,
    itmax );
```

Listing 12: Calling gradient_descent2 for the Ford data.

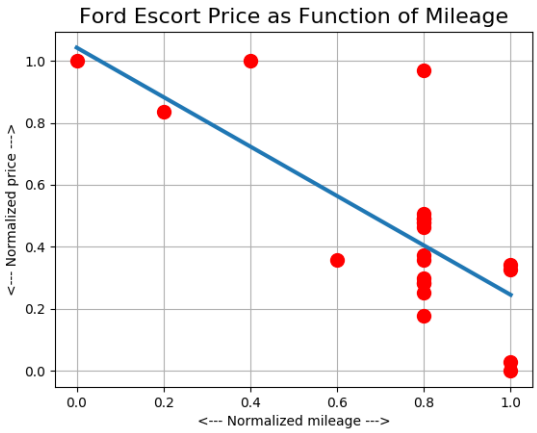
10 Example: Gradient descent to fit Ford data

We run `gradient_descent2` on the Ford problem, with a starting guess of `bm0=[b,m]=[0.5,0.0]`:

```
ford_gradient2:
  Given mileage x and price y for used Fords,
  seek (m,b) so that y=b+mx approximates the data.
  Use gradient descent2 to estimate best b and m.

it:   0 x: [0.5           0.          ] f(x) 2.5773498181818186 df(x) [-9.05163636 -1.34333243]
it:   1 x: [0.59051636    0.01343332] f(x) 1.954602113941928  df(x) [-4.61119726  0.68933256]
it:   2 x: [0.63662834    0.00654     ] f(x) 1.7798747361978844 df(x) [-2.63202845  1.55273919]
it:   3 x: [ 0.66294862 -0.00898739] f(x) 1.6955150280382623 df(x) [-1.74111328  1.90032506]
 ... ... more output ... ...
it: 117 x: [ 1.0471238  -0.78465017] f(x) 0.5783556887827288 df(x) [-0.04539271  0.09537134]
it: 118 x: [ 1.04757773 -0.78560388] f(x) 0.578245646967721  df(x) [-0.04415571  0.09277238]
it: 119 x: [ 1.04801929 -0.78653161] f(x) 0.5781415209296815 df(x) [-0.04295242  0.09024424]

  Step size ||dx|| less than dxtol =  0.001

  Norm of initial error is 2.57735
  Norm of total error is 0.578142
```

Listing 13: Output for gradient_descent2 on Ford data

Our parameters are `b=1.048, m = -0.786`. We plot the normalized data:



A linear formula $y = b + mx$ approximates the normalized Ford data

11 No Computing Assignment for this Lab!