Optimization:
Minimization by Parabolic Interpolation

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/∼jburkardt/classes/math2070_2019/optimization_parabolic/optimization_parabolic.pdf

How do we find the lowest point?

The bisection method

Given a function $f(x)$ and an interval $[a,b]$, determine an argument $a \leq x^* \leq b$ at which $f(x)$ attains a locally lowest value.

1 Minimization by Golden Search

Suppose we believe a function $f(x)$ has a minimizer x^* in the interval $[a,b]$, and we have no other special information. We should realize that we can only come within some tolerance of the exact location, so we may be willing to accept an estimate x such that $|x^* - x| <$ xtol, where `xtol` is a tolerance we specify.

The golden search algorithm provides a way of organizing the sampling of the interval so that the interval of uncertainty decreases as fast as possible. In comparison to bisection, which decreased the uncertainty interval by a factor of 0.5 each step, we will expect to see a decrease by a factor of about $\frac{2}{1+\sqrt{(5)}} = \frac{1}{\phi} \approx 0.618$. Here ϕ is a famous mathematical constant known as the *golden ratio*.

The algorithm is an iteration. Each step begins with an interval $[a,b]$. It inserts two test points `c` and `d` so that $a < c < d < b$. The spacing of these points is chosen using the golden ratio. This turns out to guarantee the best convergence rate for the algorithm.

We now do a simple test: if $f(c) < f(d)$, the right endpoint of our interval is now `d`; otherwise the left endpoint of the interval becomes `c`. This shrinks the interval to 0.618 of its previous sizem, in other words, our shrinkage factor $\alpha = 0.618$. Unless the interval is smaller than our tolerance, we now have a new interval $[a,b]$ and can take the next step of the iteration.

2 Example: Golden search on the sag() function

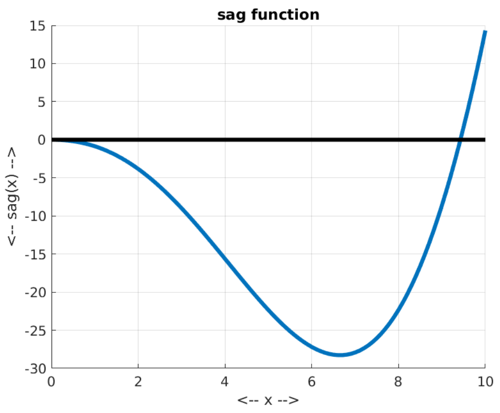
A MATLAB implementation of golden search is very simple:

```
function x = golden_search ( f, a, b, xtol )

  phi = ( sqrt ( 5.0 ) + 1.0 ) / 2.0;

  while ( true )

    c = b - ( b - a ) / phi;
    d = a + ( b - a ) / phi;

    if ( xtol < abs ( c - d ) <= xtol )
      break
    end

    if ( f ( c ) < f ( d ) )
      b = d;
    else
      a = c;
    end

  end

  x = ( b + a ) / 2.0;

  return
end
```

Listing 1: golden_search.m

We can apply the bracketing operation to the *sag()* function:

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



The sag() example function.

Starting with the interval $[0, 10]$, the golden search code finds an approximate minimizer, and `dx` decreases by a factor of $\frac{1}{\phi}$ on each step:

```
 1 f(3.819660112501052) = -27.6945  dx = 6.18034   alpha = 0.618034
 2 f(6.180339887498949) = -27.6945  dx = 3.81966   alpha = 0.618034
 3 f(5.278640450004206) = -27.6945  dx = 2.36068   alpha = 0.618034
 4 f(6.180339887498948) = -28.2536  dx = 1.45898   alpha = 0.618034
 5 f(6.737620787507360) = -28.2536  dx = 0.901699   alpha = 0.618034
..  ...  ...
34 f(6.650577017997207) = -28.2751  dx = 7.84188e-07  alpha = 0.618034
35 f(6.650577317530397) = -28.2751  dx = 4.84655e-07  alpha = 0.618034
36 f(6.650577203118899) = -28.2751  dx = 2.99533e-07  alpha = 0.618034
```

Listing 2: Minimization of sag() using golden search.

3 Searching for a bracket interval

Golden search assumes that we have an interval within which we know there is a minimizer. In general, when searching for the minimizer, we may start our search without even a good idea of where the minimizer might be. So before we look closely for the minimizer, we have to first find a good hunting ground.

Assuming that $f(x)$ is continuous, then for zero finding, knowing that two points at which $f()$ has opposite sign guarantees that a root exists, and that it lies between those two points.

To make a similar guarantee that a minimizer exists, it is necessary to have found three points, $a < b < c$, such that $f(b)$ is less than $f(a)$ and $f(c)$. If we have such data, then we are said to have *bracketed* the minimizer. Let us consider how we might search for such bracket data in case we have no information beforehand about out function.

To begin, choose some two points that are a "reasonable" distance apart.. Denote by a the point with the larger function value, and b the other point. Let $h = b - a$ and compute $c = b + h$.

If $f(c) > f(b)$, then we have succesfully bracketed a minimum in $[a, c]$, and are done. Otherwise, set `a` to the value of `b`, set `b` to the value of `c`, and compute `c = b + h` and repeat the test.

This search will fail if there is no minimizer, or if the spacing h is too large, or if the choice of a and b is unfortunate. Obviously a plot of the function would presumably be useful, but we are interested in an automatic procedure that does not require human intervention; moreover, a plot can be expensive in terms of function evaluations, and has some of the same limitations in terms of scope and scale that can allow a minimizer to be missed.

4 The bracket function

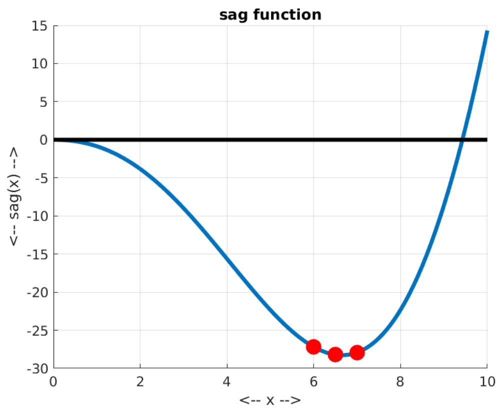
Here is a simple version of a bracketing function:

```
function [ a, b, c ] = bracket ( f, x1, x2 )

  if ( f(x1) > f(x2) )
    a = x1;
    b = x2;
  else
    a = x2;
    b = x1;
  end
  c = b + ( b - a );

  itmax = 100;
  it = 0;
  while ( f(c) < f(b) && it < itmax )
    a = b;
    b = c;
    c = b + ( b - a );
    it = it + 1;
    if ( itmax <= it )
      fprintf ( 1, 'bracket failed, exceeded iteration limit %d\n', itmax );
      return
    end
  end

  return
end
```

Listing 3: The bracket function

5 Example: bracketing the sag() function

We can apply the bracketing operation to the *sag()* function. Using a stepsize of 0.5, and starting at 2, we easily find three bracketing points.



Three points that bracket a minimizer

```
bracket seeks points a, b, c which bracket a minimizer
of a function f(x) so that f(a) > f(b) < f(c).

Search begins with:
x1: f(2) = -3.8382
x2: f(2.5) = -6.16977

bracket returns (a,b,c):

a: f(6) = -27.1928
b: f(6.5) = -28.2128
c: f(7) = -27.9175
```

Listing 4: Output from bracketing sag().

6 Exercise: bracketing the hump() function

Consider the function *hump.m*. You may already have a copy of this in your directory. Otherwise, download it from the web page for today's lab. Recall that this function has two peaks between 0 and 10, and then seems to decrease outside that interval.

Try the following command:

```
[a,b,c] = bracket ( @(x)hump(x), 0.0, 1.0 );
```

Luckily, the code won't take more than 100 steps, or we would have been in real trouble. In this case, the bracket search moves to the left, into negative values, because $f(0) < f(1)$. We were hoping to focus on the minimizer that lies to the right, instead. We can retry this operation:

```
[a,b,c] = bracket ( @(x)hump(x), 1.0, 2.0 );
```

but again the bracketing search simply runs to the right without finding a good result. Remember, the code is only dealing with a sample of the *hump()* function, so if the stepsize is not small enough, it could miss important facts. Let's try

```
[a,b,c] = bracket ( @(x)hump(x), 0.4, 0.5 );
```

Now the bracket code is able to detect a "valley" in the sampled data.

Now use a and c as your bracket interval, and call *golden_search()* to find the location of the minimizer.

7 Computing and minimizing the parabolic interpolant to data

Now suppose we have bracketed a minimizer with points a, b, c. We can now try to create a model of the data $(a, f(a)), (b, f(b)), (c, f(c))$. That is, we can model the data with a parabola, whose minimum will be easily obtained. The formula for the necesary parabolic interpolant is easily written using a Lagrange basis:

$$p(x) = f(a)\frac{(x-b)(x-c)}{(a-b)(a-c)} + f(b)\frac{(x-a)(x-c)}{(a-b)(a-c)} + f(c)\frac{(x-a)(x-b)}{(c-a)(c-b)}$$

The derivative is

$$p'(x) = f(a)\frac{(x-b)+(x-c)}{(a-b)(a-c)} + f(b)\frac{(x-a)+(x-c)}{(a-b)(a-c)} + f(c)\frac{(x-a)+(x-b)}{(c-a)(c-b)}$$

and the point at which $p'(x) = 0$ is

$$x = \frac{1}{2}\frac{(b^2-a^2)(f(b)-f(c)) - (b^2-c^2)(f(b)-f(c))}{(b-a)(f(b)-f(c)) - (b-c)(f(b)-f(c))}$$

8 Repeated parabolic interpolation

If the parabolic model was a good approximation to the function, then x, the minimizer of $p(x)$, should be a good approximation to x^*, the minimizer of $f(x)$. However, we can try to improve the approximation. We do this by replacing the oldest point in our bracketing set by the value x. Symbolically,

$$[a, b, c] \leftarrow [b, c, x]$$

We can then construct the parabolic interpolant to this new set of points, and repeat this operation until we see little change in the results.

Assume that we have a function *x=parabola_critical(@(x)f(x),a,b,c)* which returns the critical point x of the parabola through $(a, f(a)), (b, f(b)), (c, f(c))$. Such a code might look like this:

```
function x = parabola_critical ( f, a, b, c )

  top = ( b - a ) * ( b + a ) * ( f(b) - f(c) ) ...
      - ( b - c ) * ( b + c ) * ( f(b) - f(a) );

  bot = ( b - a ) * ( f(b) - f(c) ) ...
      - ( b - c ) * ( f(b) - f(a) );

  x = 0.5 * top / bot;

  return
end
```

Listing 5: 'Code to find critical point of a parabola.

Then our minimizing code is simply:

```
function x = parabolic_minimizer ( f, a, b, c, xtol )

  while ( true )

    abc_min = min ( [ a, b, c ] );
    abc_max = max ( [ a, b, c ] );
    dx = abc_max - abc_min;
    if ( dx < xtol )
      return
    end

    x = parabola_critical ( f, a, b, c );

    a = b;
    b = c;
    c = x;

  end

  return
end
```

Listing 6: Minimization by repeated parabolic iterpolation

9 Example: Minimizing the sag() function

If we run the code *parabolic_minimizer_test.m*, and turn on some internal print statements, we see:

```
  Search begins with:
  a: f(0) = -0
  b: f(5) = -22.35
  c: f(10) = 14.331
it=1  dx=10  x=     4.393071959  f(x)=  -18.36219159531186
it=2  dx=5.60693  x=     6.021086214  f(x)=  -27.25859975151059
it=3  dx=5.60693  x=     6.169550648  f(x)=  -27.66850875323199
it=4  dx=1.77648  x=     7.002418071  f(x)=  -27.91238836170857
it=5  dx=0.981332  x=     6.644196034  f(x)=   -28.2749659619705
...
it=11  dx=5.23685e-06  x=     6.650577264  f(x)=  -28.27507993917309
it=12  dx=2.84806e-07  x=     6.650577327  f(x)=  -28.27507993917307

  parabolic_minimizer returns estimated minimizer x:

  x: f(6.65058) = -28.2751
```

Listing 7: Parabolic minimization of sag().

and it's clear that parabolic minimization reached the minimizer much faster than golden search did.

10 Assignment #6

Over the interval $0 \le x \le 7$, the function

$$f(x) = \cos(x) + 5\cos(1.6x) - 2\cos(2x) + 5\cos(4.5x) + 7\cos(9x)$$

seems to have about 10 local minima. Use golden search or parabolic interpolation to estimate the values of x at which $f(x)$ attains its two lowest local minima in this interval. A plot may help you get started.

Turn in: your file *hw6.m* by Friday, October 4. It should print out the starting interval used, compute the minimizer x, and print the minimizer's function value $f(x)$ for both cases.