Numerical Solution to the Brachistochrone Problem

Aditya Mittal*
Stanford University

Abstract

This paper presents the numerical solution to the brachistochrone problem. The brachistochrone problem is one of the earliest problems in calculus of variations and has been solved analytically by many including Leibniz, L'Hospital, Newton, and the two Bernoullis. It asks the question "What is the shortest time path a particle can take from point a to point b given that it starts at rest and is accelerated by gravity without friction?" The analytical solution to the brachistochrone problem is a segment of the cycloid, which is the curve defined by a point on the circumference of a circular disk rolling on a flat surface. In this paper I present the computation of this segment of the cycloid as the solution to a nonconvex numerical optimization problem.

Introduction to the brachistochrone problem

The brachistochrone problem has a well known analytical solution that is easily computed using basic principles in physics and calculus. The solution is a segment of the curve known as the cycloid, which shows that the particle at some point may actually travel uphill, but is still faster than any other path. Let us now compute this solution analytically. The time to travel from point a to point b is the integral of the arc length over the speed from a to b:

$$t_{ab} = \int_a^b \frac{ds}{v}$$

By conservation of energy the speed at any point of the trajectory is computed by equating kinetic and potential energies and solving for the speed v:

$$\frac{1}{2}mv^2 = mgy$$

$$v = \sqrt{2gy}$$

Next we use this velocity and the identity $ds = \sqrt{dx^2 + dy^2} = \sqrt{(1 + y'^2)}dx$ to get the time from a to b:

$$t_{ab} = \int_a^b \sqrt{\frac{(1 + y'^2)}{2gy}}dx = \int_a^b f dx$$

Here f is the function to be varied in order to compute the fastest time for this analytical solution. Since this function is independent of the x variable, and so the partial of the function with respect to x is 0, Beltrami's identity can be used to compute the partial of the function with respect to y':

Aditya Mittal is an MS student in Computational and Mathematical Engineering at Stanford University. This is a project report for CME304/MS&E315 course taught by professor Walter Murray. Contact: admittal@stanford.edu*

$$Beltrami's\ Identity: f - \frac{y'\partial f}{\partial y'} = C$$

$$\frac{\partial f}{\partial y'} = y'\left(1 + y'^2\right)^{-\frac{1}{2}}(2gy)^{-\frac{1}{2}}$$

$$f - y'\frac{\partial f}{\partial y'} = f - y'^2\left(1 + y'^2\right)^{-\frac{1}{2}}(2gy)^{-\frac{1}{2}} = C$$

Simplifying we get:

$$\frac{1}{\sqrt{(2gy)(1 + y'^2)}} = C$$

Square both sides and rearrange to get:

$$\left[1 + \left(\frac{dy}{dx}\right)^2\right]y = \frac{1}{2gC^2} = k^2$$

Here the square of the constant C has been expressed in terms of a new positive constant k^2 and this equation can be solved by the parametric equations of the cycloid which are:

$$x = \frac{1}{2}k^2(\theta - sin\theta)$$

$$y = \frac{1}{2}k^2(1 - \cos\theta)$$

Now that we know how this equation is solved analytically, let us proceed to confirm our answer numerically. The approach to solving the brachistochrone problem numerically will be to first determine the function to optimize, then to discretize this function and finally to optimize the discretization using the Quasi-Newton method. The linesearch method is used in the Quasi-Newton using the BFGS update. The Matlab code for the problem is given in the Appendix 1.

The problem setup

The idea behind solving this problem numerically is to take the time function and minimize it using a linesearch method since we are trying to find the solution that takes the least amount of time. The issues involved are what form of discretization should be used, what should be the termination criterion for the linesearch, and what should be the initial estimate. The problem will start from some point (0,1) and go to some point (1,a) where a is chosen to allow us to see different curves. The finer the discretization gets, the closer we should get to the cycloid curve.

The form of discretization to use for time function and it's gradient

There are many possibilities for how the discretization is done. After a lot of trial and error here is what I finally use. Start with the time function represented as the integral presented in the analytical solution:

$$t_{ab} = \int_a^b \sqrt{\frac{(1+y'^2)}{2gy}}\,dx$$

And remove the $\sqrt{2g}$ factor because it some constant and can be normalized by letting $g = 1/2$ and will have no effect on the actual shape and nature of the solution and then let us discretize this. In order to do this discretization let us first divide the domain from 0 to 1 into n equal regions, and approximate the solution curve by a line segment within each region so that as $n \to \infty$ the discretization approaches the actual solution curve. This kind of discretization actually works pretty well. Some of the things I tried earlier did not include treating each piece of the discretization as a line segment but rather the curve itself, in which case to get somewhat correct curves the grid (the n regions on the domain) have to be spaced such that when the gradient of the solution is higher, the spacing must be finer in order to accommodate the larger change in y that occurs. While this works most of the time, it sometimes fails to converge to a good local minimum in which case the curve looks less like a cycloid even with large n. Rather than getting into the details of the methods that didn't work so well and have to be adjusted in many such ways let me just present the one that does work without much trouble, i.e. n evenly spaced regions on the domain and the curve approximated by a line segment on each region. So, now for this discretization on each region the derivative of the curve is simply the slope of the line:

$$y_i' = \frac{dy_i}{dx_i}$$

So for the particle to slide from (0,1) to (1,0) the time taken is:

$$t_{ab} = \int_0^1 \sqrt{\frac{(1+y'^2)}{y(x)}}\,dx = \sum_{i=1}^{n} \int_{x_i-1}^{x_i} \sqrt{\frac{\left(1+\left(\frac{dy_i}{dx_i}\right)^2\right)}{y(x)}}\,dx = \sum_{i=1}^{n} \sqrt{\left(1+\left(\frac{dy_i}{dx_i}\right)^2\right)} \int_{x_i-1}^{x_i} \sqrt{\frac{1}{y(x)}}\,dx$$

here $y(x) = \frac{dy_i}{dx_i}(x - x_i) + y_i$ by the equation of a line so the above becomes

$$= \sum_{i=1}^{n} \sqrt{\left(1+\left(\frac{dy_i}{dx_i}\right)^2\right)} \int_{x_i-1}^{x_i} \sqrt{\frac{1}{\frac{dy_i}{dx_i}(x - x_i) + y_i}}\,dx$$

$$= \sum_{i=1}^{n} \sqrt{\left(1+\left(\frac{dy_i}{dx_i}\right)^2\right)} \left[\left(\frac{dy_i}{dx_i}(x - x_i) + y_i\right)^{\frac{1}{2}} \times \frac{2dx_i}{dy_i}\right]_{x_{i-1}}^{x_i}$$

$$= \sum_{i=1}^{n} \sqrt{\left(1 + \left(\frac{dy_i}{dx_i}\right)^2\right)} \times \frac{2dx_i}{dy_i} \left[\left(\frac{dy_i}{dx_i}(x_i - x_i) + y_i\right)^{\frac{1}{2}} - \left(\frac{dy_i}{dx_i}(x_{i-1} - x_i) + y_i\right)^{\frac{1}{2}} \right]$$

Define $\Delta x_i = (x_i - x_{i-1})$ and $\Delta y_i = (y_i - y_{i-1})$ so that $\frac{dy_i}{dx_i} = \frac{\Delta y_i}{\Delta x_i}$

$$= \sum_{i=1}^{n} \sqrt{\left(1 + \left(\frac{dy_i}{dx_i}\right)^2\right)} \times \frac{2dx_i}{dy_i} \left[\left(\frac{dy_i}{dx_i}(0) + y_i\right)^{\frac{1}{2}} - \left(\frac{\Delta y_i}{\Delta x_i}(-\Delta x_i) + y_i\right)^{\frac{1}{2}} \right]$$

So with some simple cancelling and rearranging this simplifies to:

$$= \sum_{i=1}^{n} 2 \left(\frac{\sqrt{\Delta x^2 + \Delta y^2}}{\sqrt{y_i} + \sqrt{y_{i-1}}} \right)$$

This defines the discretization I use to define my time function in the Matlab file t.m. Next we need a discretization for its gradient because the linesearch will require as input the discretization of the gradient as well. The discretization for the gradient is actually quite easy since we can just call the time function at each of the n points in the discretization by adding a small dx and by subtracting a small dx and then computing the slope at that point by taking the change in y over the change in x around that point. This is basically the finite difference technique for approximating the derivative at a given point.

The Quasi-Newton method with BFGS update and the linesearch algorithm

The linesearch algorithm is an iterative approach for finding the step length α_k used to find the local minimum x^* of an objective function $f: R^n \to R$. In this problem I conduct a linesearch and use the quasi-Newton method which is a modification of the Newton method for the case when the gradient is 0. The Newton method does not work in the case when the gradient is 0 because the Newton method assumes that the function can be locally approximated as a quadratic in the region around the optimum, and uses the first and second derivatives of the function to find the stationary point. The Newton iterate to find a stationary point is given by:

$$x_{k+1} = x_k - \frac{f(x)}{f'(x)}$$

So by finding the stationary points of the derivative one can find local minimizer or maximize of the function through an iterative process. However, in our case, the problem is not convex and therefore the gradient of the function can be 0 and also it is expensive to numerically compute the Hessian of the function. The BFGS method suggested by Broyden, Fletcher, Goldfarb, and Shanno in 1970 provides a method to approximate the Hessian as:

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k (B_k s_k)^T}{s_k^T B_k s_k}$$

where $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ and s_k is obtained by solving $B_k s_k = -\nabla f(x_k)$. $f(x)$ of course denotes the function to be minimized and $x_{k+1} = x_k + \alpha_k s_k$ is the Newton step, where the proper step length α_k is found by using the linesearch method.

The linesearch method as already mentioned is used to find α_k and to do this I use the Wolfe conditions to do an "inexact" linesearch because it is computationally much cheaper than an "exact" linesearch which requires minimizing α exactly over the whole real line. There are two Wolfe conditions, one is the Goldstein condition and the other is the curvature condition. The Goldstein condition ensures that α_k decreases sufficiently and the curvature condition ensures that the slope of the function $\phi(\alpha) = f(x_k + \alpha p_k)$ at α_k is greater than some given constant c_2 at $\alpha = 0$. For my linesearch I use the following two "strong" Wolfe conditions to ensure that α_k lies close to the critical point:

$$Goldstein\ condition: f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k p_k^T \nabla f(x_k)$$

$$Curvature\ condition: \left| p_k^T \nabla f(x_k + \alpha_k p_k) \right| \leq c_2 \left| p_k^T \nabla f(x_k) \right|$$

So as long as the linesearch can find an alpha that satisfies these two conditions we have an alpha that is good for the step length. So, how do I find such an alpha? Well, we want to find it as quickly as possible without spending too many iterations looking for an alpha, so I begin with the conjecture that an exponential function will be fast enough to get to it and starting from $\alpha = 0$, I iterate $\alpha = \alpha + .01e^{\alpha}$ until I find an alpha that satisfies both the Wolfe conditions. Notice that as I get farther and farther the step length I take to look for an alpha increases. In worst case, the conditions will be satisfied in a small interval far away from 0, in which case this method will skip over that interval and never find the interval in which the Wolfe conditions are satisfied. In this case, however, I observe that taking small steps such that the α does not miss the interval will require too many iterations anyway and I might as well repeat with a smaller coefficient for the exponential if my program outputs an alpha termination error so this method works extremely well.

Now that I have the step length, all I need is the decent direction to compute the next iterate and this is easily computed using the BFGS method's approximation of the Hessian given earlier as

$$p_k = -B_k / \nabla f(x_k)$$

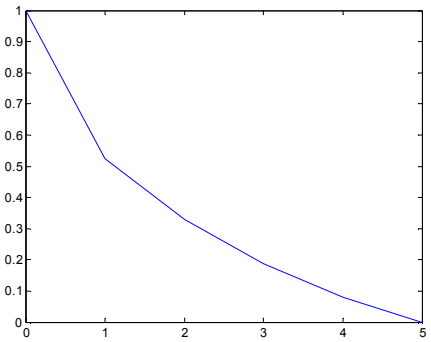
Lastly, I need some termination criterion to know when to stop these iterations and for this I simply set some threshold value to determine when the slope of the function is close enough to 0 because then we have found a stationary point $\|\nabla f(x_k)\| < \beta$ where $\beta = .0001$ is the threshold value I use.

My linesearch and discretization is really good so just starting with any reasonable initial curve works fine and I can start with a straight line between the two points or a parabola or anything else that is reasonable. For simplicity I just start with a straight line and have a plotfunction.m which takes in the number of regions to discretize into and the height $0 \leq a < 1$ to start with and provides the curve.

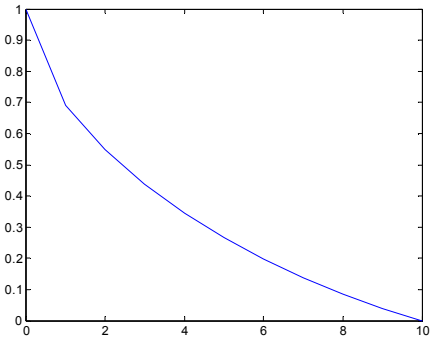
Resulting graphs

Here are the resulting plots showing the cycloid solution. The number of points in the discretization is n, and the height at x=1 is a. All the graphs start with height 1 and fall to height a. For most of the graphs I let a = 0, and vary n. For n = 20, I show graphs with all the iterations instead of just the final iteration so

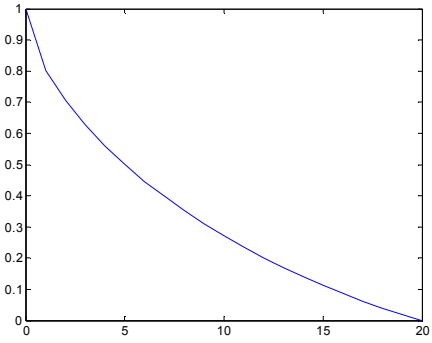
we can visualize how the convergence is occurring, and I also show a graph where I let a = .4 and then a graph with a = .8 to demonstrate that the shortest time curve may also go upwards. Any desired case can easily be run using the code given in the Appendix 1 by simply calling the plotsolution function with the desired values for n and a.



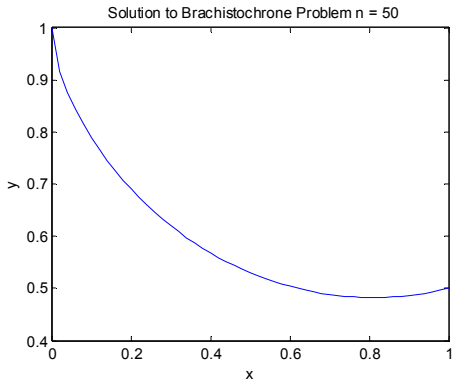
Graph for n=5 and a=0.



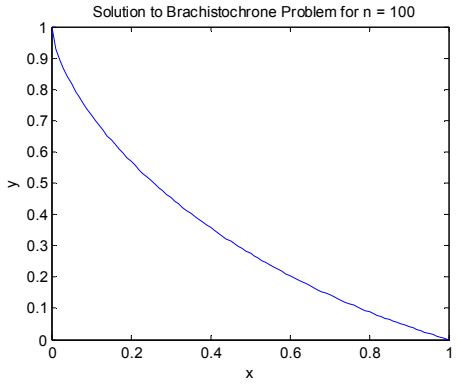
Graph for n=10 and a=0



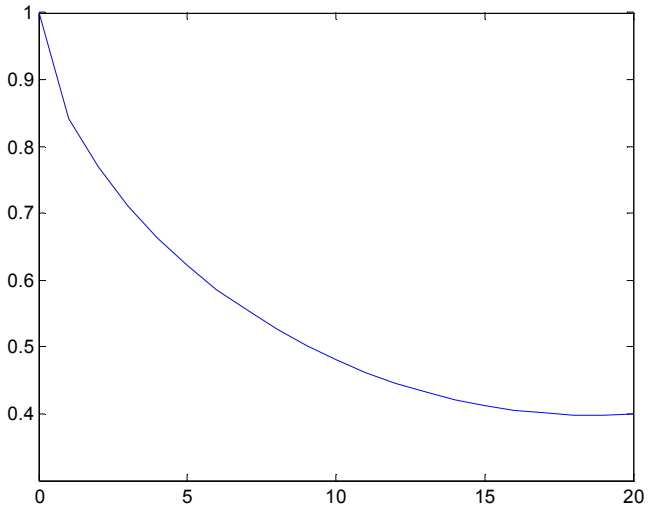
Graph for n=20 and a=0



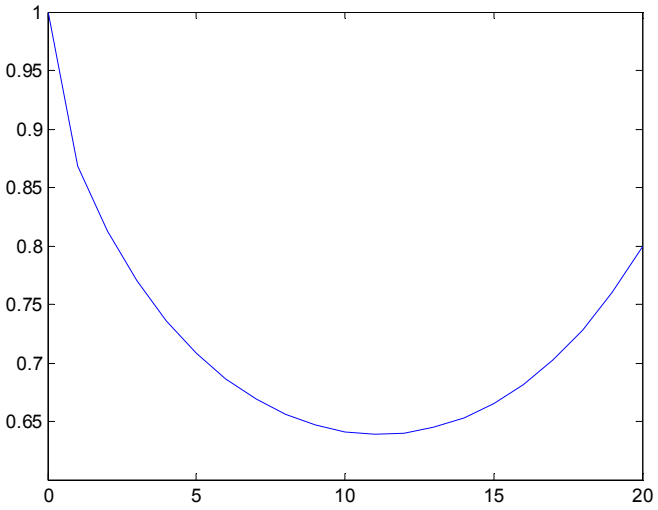
Graph for n=50 getting pretty smooth with a=.5 just a small period of time in the end when the particle travels upwards



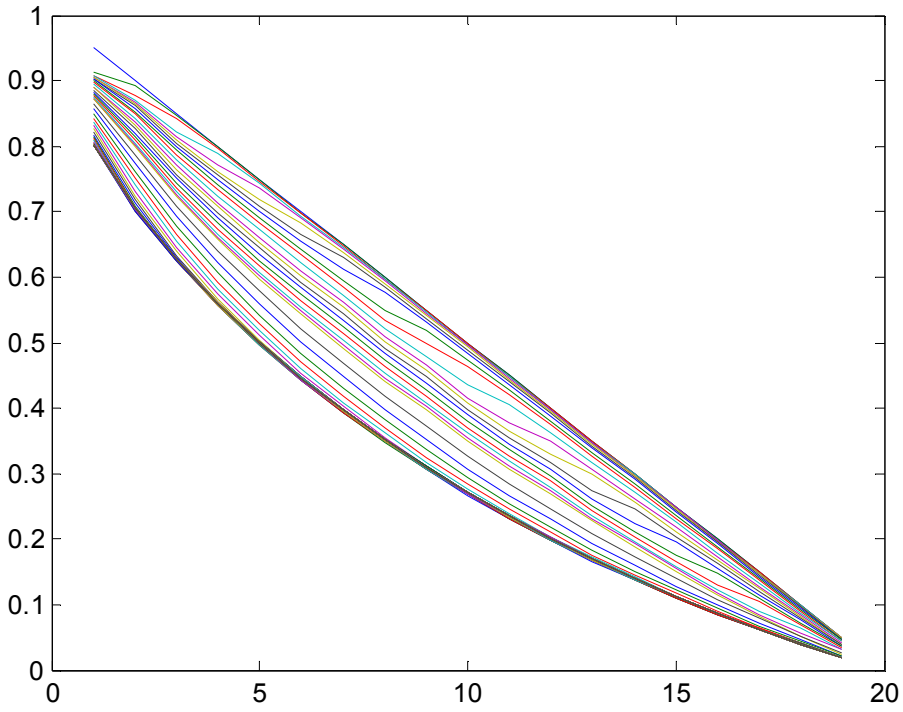
Graph for n=100 with a=0, getting very fine not much more visible notches in the graph



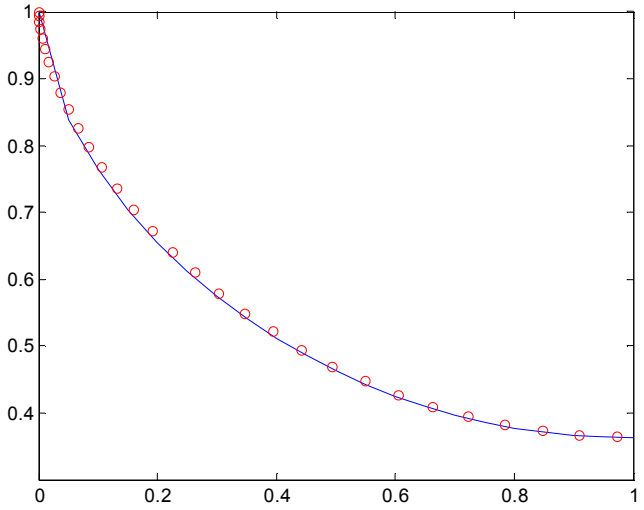
Graph for n=20 with a=0.4 instead of a=0 like the other graphs, still a cycloid



The Graph for n=20 and a =0.8, showing that the shortest time path may be such that it goes down and then comes back up



Graph showing the 20 iterations starting from a straight line from (0,1) to (1,0) to the cycloid without the boundary values appended to the graph



Here the circles are plotted using the equation of a cycloid, and the line is plotted as usual for n=20 and a=.3634 just to show that the program indeed is outputting accurate results. The equation of the cycloid in this instance is $x = a(b - \sin(b))$ and $y = 1 - a(1 - \cos(b))$ where $a = \frac{1}{\pi}$ and $b = \pi$.

Conclusion

The numerical solution to the Brachistochrone problem works out pretty neatly and it was quite interesting to see the accuracy of the resulting graphs for different n. First I had tried steepest descent which obviously did not work because the problem in not convex and I discovered that Quasi-Newton with the BFGS method works really well. The program runs quite quickly and provides reasonably good results within around 20 iterations, unless we run it for large values of n such as 100 to get very fine precision of course. I got lucky with my choice of alpha since it just worked the first time through in my linesearch and continued to work for all cases, in some sense one could call it a forward tracking method for finding alpha. Of course there is no end to playing with the parameters, measuring errors relative to the actual cycloid curve, getting more and more graphs for different cases, looking at gradients or contour plots and analyzing them, tabulating number of iterations and running times and computing big O or flops, making more variety of initial guesses etc. but it works well and I've learnt and demonstrated the optimization methods necessary to solve such a problem so I will stop here.

References

[1] Class notes from CME304/MS&E315 taught by Professor Walter Murray, Stanford University.

[2] Nocedal, Jorge, and Stephen J. Wright. Numerical Optimization. Springer, 2000.

[3] http://mathworld.wolfram.com/BrachistochroneProblem.html. April 2008.

[4] http://mathworld.wolfram.com/Cycloid.html. April 2008.

[5] http://en.wikipedia.org/wiki/Brachistochrone. April 2008.

[6] http://en.wikipedia.org/wiki/Quasi-Newton_method. April 2008.

[7] http://en.wikipedia.org/wiki/Wolfe_conditions. April 2008.

[8] http://en.wikipedia.org/wiki/BFGS_method. April 2008.

[9] http://en.wikipedia.org/wiki/Finite_difference. April 2008.

Acknowledgements

I would like to thank Professor Walter Murray for his enlightening lectures and great selection of problem sets for us to do, and also Nick Henderson and Kathy Laura Jenson for being the best TA's and helping us learn a lot of optimization in such a short period of time.

Appendix 1: Matlab code

Linesearch.m

```
%Linesearch or Steplength method by Aditya Mittal
%Iterative Approach to find local minimum of an objective function
%Uses the Quasi-Newton method and Wolfe conditions
function [mini] = ...
   linesearch(func, grad, guess)% tol, alpha_M)
%Given the function handle: func
%Given the gradient handle: grad
%Given an initial guess for the minimum: guess
%Given a tolerance: tol
%Given a upper bound on the permitted step: alpha_M
%Outputs the computed local minimum: mini
%While \norm(grad(x[k]))>tol:
%Compute the descent direction p[k]
%Choose alpha[k] to 'loosely' minimize phi_alpha=func(x[k]+alpha*p[k])
%Update x[k+1]=x[k]+alpha[k]*p[k], k=k+1
tic;
k=1;
x(:,1)=guess;
[m,n]=size(grad(x(:,1)));
B = eye(m);
if(n~=1)
   disp 'Gradient needs to be a column vector'
   return
end
while norm(grad(x(:,k)))>.0001
   p(:,k)=-B\grad(x(:,k));
   while p(:,k)'*grad(x(:,k))>=0
```

```
    %Try a different descent direction p(k)
    %Doing Quasi Newton with BFGS update
    % s(:,k)=x(:,k+1)-x(:,k);
    % y(:,k)=grad(x(:,k+1))-grad(x(:,k));
    % B=B-(B*s(:,k)*s(:,k)'*B)/(s(:,k)'*B*s(:,k))+(y(:,k)*y(:,k)')/(y(:,k)'*s(:,k))
    % p(:,k)=-B(:,k)\grad(x(:,k));
     disp 'descent direction compute error'
  end
  %Choose alpha(k) to satisfy Wolfe conditions
  %Choose constants c and d such that 0<c<d<1
  c=.3;
  d=.7;
  alpha= 0;
  count = 0;
  % Goldstein condition followed by curvature condition
  while (func(x(:,k)+alpha*p(:,k))>func(x(:,k))+c*alpha*p(:,k)'*grad(x(:,k))...
        || abs(p(:,k)'*grad(x(:,k)+alpha*p(:,k)))>d*abs(p(:,k)'*grad(x(:,k))))
    %Choose new alpha
    %alpha_inc = .1*exp(alpha);%My method of quickly computing the increment in alpha
    alpha_inc = .01*exp(alpha);%My method of quickly computing the increment in alpha
    alpha = alpha+alpha_inc;
    count = count+1;
    if count == 10000
      disp 'alpha termination error'
      return
    end
  end
  %Update
  x(:,k+1)=x(:,k)+alpha*p(:,k);
  s(:,k)=x(:,k+1)-x(:,k);
  y(:,k)=grad(x(:,k+1))-grad(x(:,k));
  B=B-(B*s(:,k)*s(:,k)'*B)/(s(:,k)'*B*s(:,k))+(y(:,k)*y(:,k)')/(y(:,k)'*s(:,k));
  k=k+1;
end
mini = x;
toc
```

g.m (The Gradient of Time Function)

```
function g = g(y, y0, yn)
n = size(y, 1) + 1;
g = zeros(n-1, 1);
for i = 1:n-1
  yp = y;
  ym = y;
  yp(i) = y(i) * (1+10^(-7));
  ym(i) = y(i) * (1-10^(-7));
  g(i) = (t(yp, y0, yn) - t(ym, y0, yn)) / (2*10^(-7)*abs(y(i)));
```

```
end
```

t.m (The Time Function)

```
function value = t(y, y0, yn)
n = size(y, 1) + 1;
value = 0;
value = value + 2 * sqrt(1/n^2 + (y(1) - y0)^2) / (sqrt(y(1)) + sqrt(y0));
value = value + 2 * sqrt(1/n^2 + (yn - y(n-1))^2) / (sqrt(yn) + sqrt(y(n-1)));
for i = 2 : n-1
   value = value + 2 * sqrt(1/n^2 + (y(i) - y(i-1))^2) / (sqrt(y(i)) + sqrt(y(i-1)));
end
for i=1:n-1
   o(i) = sum(q(1:i,a));
end
plot([0;o;1],[1;1-r(:,a);0])
```

plotsolution.m

```
function x=plotsolution(n,a)
%a is the final height, must be greater than 0 and less than 1
%n is the number of discrete regions for the discretization
if (a < 0 || a >=1)
   disp height a must be less than 1 and greater than or equal to 0
end
y0=0;%y points down so this in Cartesian system corresponds to (0,1)
yn=1-a;%y points down so this in Cartesian system corresponds to (1,a)
y=[1/n:1/n:1-1/n]';
fh=@(y,dx)t(y,y0,yn);
gh=@(y,dx)g(y,y0,yn);
x=linesearch(fh,gh,y);
x=1-x;
[q,r]=size(x);
%append the boundary conditions to the final iteration to plot
x=[1;x(:,r);a]
plot(0:1:n,x)
```

```
%cycloid for a=.3634 (optional code to attach at the end of plotsolution.m)
a=1/3.14159265359;
b=[0:.1:3.14159265359];
x1=a*(b-sin(b));
y1=1-a*(1-cos(b));
plot(0:1/n:1,x,'-',x1,y1,'ro')
x1=1/3.14159265359*(3.14159265359-sin(3.14159265359))
y1=1-1/3.14159265359*(1-cos(3.14159265359))
```