Michael Morse
Bridge Project
MTH 437

I began my bridge process by making a simple 1 weighted system. This system evolved to contain all points. Instead of using a linear morph I morphed different aspects of my sudobridge using a linear theme till the bridge was the same as the real bridge. The table below summarizes my results. Results were calculated in 9.81 m/s/s gravity and then converted over by scaling to 1.53 m/s/s.

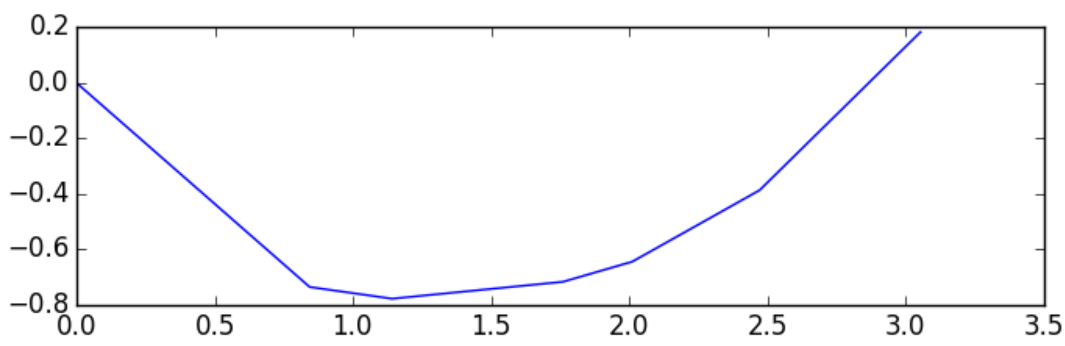
P	x coordinate	y coordinate
1	0.84328699	-0.73706652
2	1.14131493	-0.77926063
3	1.75931642	-0.71830849
4	2.01000149	-0.64565869
5	2.47062908	-0.38757686

Tension	g = 9.81 m/s/s	g = 1.53 m/s/s
T0	3.959602	0.617552605504587
T1	3.0110531	0.469613786238532
T2	2.99578741	0.467232898807339
T3	3.1039945	0.484109233944954
T4	3.41737701	0.532985405229358
T5	4.17733577	0.651511083394495

The system I used started off with a triangular system where all the hypotenuses were sqrt(2) such that y distance and x distance were simply integers. Once I used this to coverage to a starting guess, only having the center mass on, I started the morph. The first way I edited the nodes was by adding the masses back in a linear morph fashion using E5 steps. Once I had a system with all 5 masses that converged (and I had checked it) I went on to modify other points.

Again I used the linear system to bring my y end node from 0 to the actual value. At this time I realized that the nodes had to be done in a special order else convergence would not happen. Next I changed the L1 from sqrt(2) to its actual point and did the same for L2 and L5. Here I ran into some issues as I could do no more modifications without the system failing. I ended up having to modify the x end node and the L4 node at the same time in order to get convergence. Finally I had to change the last two inner nodes L3, L4. They were changed in a same fashion and converged fast.

I am pleased with my results:



Background:

On the 2nd of October we were assigned to model a bridge in equilibrium with no more data then the lengths of the wires and the end point nodes. This left a 16 dimensional problem to be solved by a quasi-Newton's method. Luckily we did have 16 independent scalar equations that, while not linear, could be used to solve the problem.

<u>6 Equations</u>
6 equations were in the form:

$$\left\|\vec{P}_{i+1}-\vec{P}_i\right\|_2 = L_i$$

Which can be changed to

$$\left\|\vec{P}_{i+1}-\vec{P}_i\right\|_2 - L_i = 0$$

or expressed as a scalar equation of the form:

$$\sqrt{\left(p_{i+1_x}-p_{i_x}\right)^2+\left(p_{i+1_y}-p_{i_y}\right)^2}-L_i=0$$

The other 10 equations came from:

$$T_{i-1}\left(\frac{\left(\vec{P}_{i-1}-\vec{P}_i\right)}{L_{i-1}}\right)+T_i\left(\frac{\left(\vec{P}_{i+1}-\vec{P}_i\right)}{L_i}\right)+m_i g\begin{pmatrix}0\\-1\end{pmatrix}=0$$

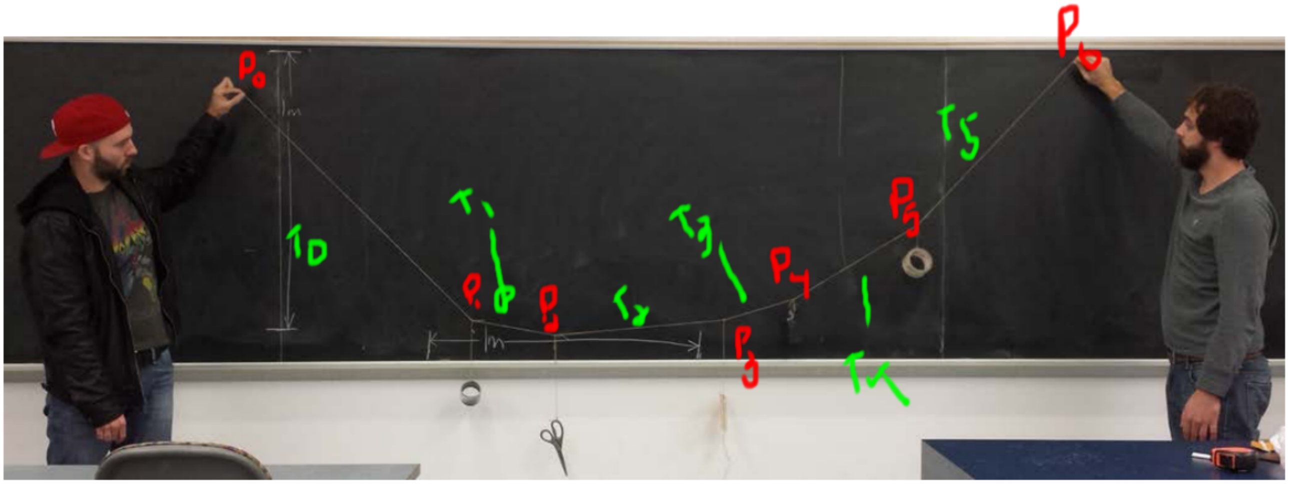
Again these can be reduced to scalar equations as:

$$T_{i-1}\left(\frac{\left(p_{i-1_x}-p_{i_x}\right)}{L_{i-1}}\right)+T_i\left(\frac{\left(p_{i+1_x}-p_{i_x}\right)}{L_i}\right)=0$$

and

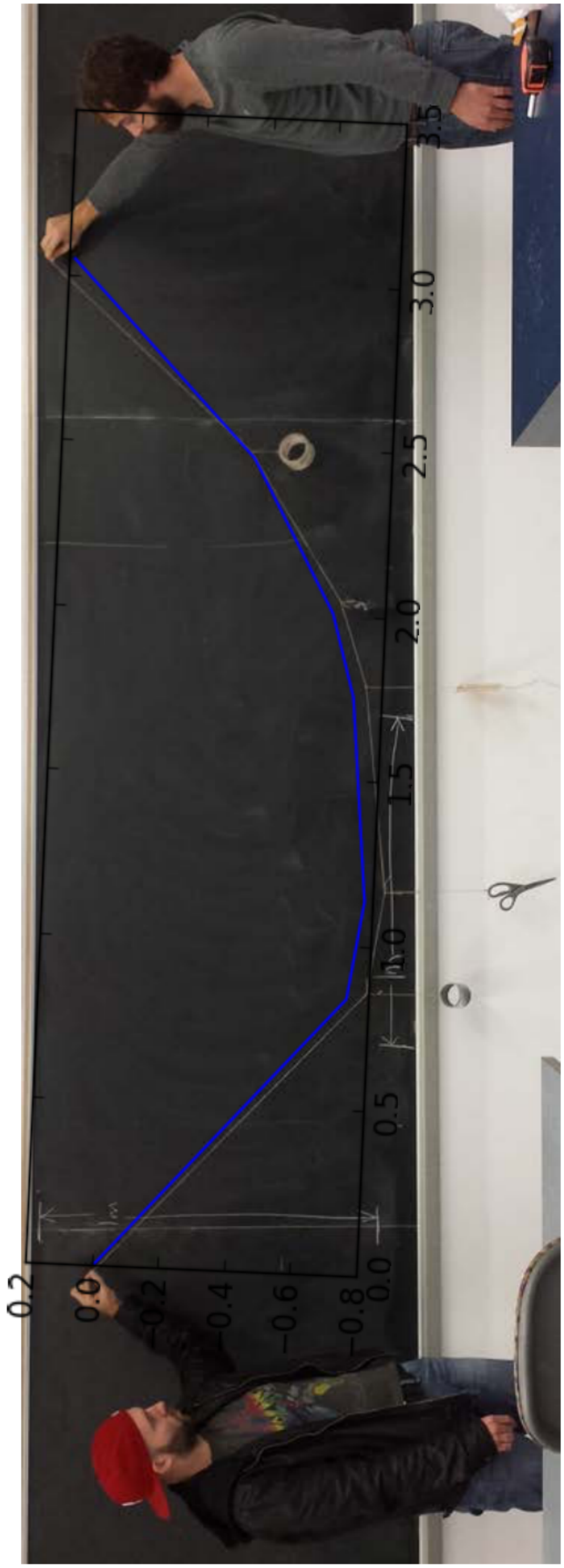
$$T_{i-1}\left(\frac{\left(p_{i-1_y}-p_{i_y}\right)}{L_{i-1}}\right)+T_i\left(\frac{\left(p_{i+1_y}-p_{i_y}\right)}{L_i}\right)-m_i g=0$$

It is also nice sometimes to be able to see what is going on, so below is a marked up picture of the bridge in order to identify the nodal points.



Superimposed

Finally to check how well my result did against the actual photo I super imposed it on top using SketchUp. The photo can be seen below. The photo shows that my solution was a little off but over all it followed the shape and lengths of the bridge. I am overall pretty satisfied with my solution.


0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.2
0.0
−0.2
−0.4
−0.6
−0.8

Codes:

This is the code for my 3rd attempt at a sudo bridge. Here is where I decided to go with the more abstract morphing technique.

```
#sudo bridge 3
from numpy import *
import broyden2
from time import sleep
import newton_multid as new

def bridge(x):

    h = 1.
    s2 = (2.)**(1./2)
    t1 = 0.806047
    L = array([s2, s2, s2, s2, s2, s2])
    M = array([h*.2226, h*0.073, .0581, h*.0822, h*.128])
    P = zeros((7,2))
    P[0] = [0,0]
    P[len(P)-1] = [6.0, 0.0]
    T = x[len(P)+3:]
    gv = array([0,-1])
    f = zeros(len(x))
    g = 9.81
    #print T
    for k in range(len(P)-2):
        P[k+1] = [x[2*k] ,x[2*k+1]]
    #print P
    for i in range(len(P)-1):
        Pd = P[i+1] - P[i]
        f[i] = linalg.norm(Pd) - L[i]
    for j in range(1,len(P)-1):
        f[i+1+2*(j-1)] = T[j-1]*(P[j-1,0]-P[j,0])/L[j-1] + T[j]*(P[j+1,0]-P[j,0])/L[j]
        f[i+2+2*(j-1)] = T[j-1]*(P[j-1,1]-P[j,1])/L[j-1] + T[j]*(P[j+1,1]-P[j,1])/L[j] - M[j-1]*9.81
        #print i+2+2*(j-1)
        #print f
    return f
x0 = array([ 0.49371527,-1.32523403,1.57620047,-2.23530315,2.95255255,-2.56034921,
  4.32614317,-2.2238246,5.3639315,-1.26309812,3.39307124, 1.54756028,
  1.21713849,  1.21958542,  1.61421268,  2.63369606])
print bridge(x0)


#print bridge(x0)
```

```
b0 = broyden2.fdjac(bridge,x0)
xnew = broyden2.broyden(x0,bridge,b0,.00001,1000)
print xnew
'''
'''
x = array([1.,-1., 2.,-2., 3.,-3., 4.,-2., 5.,-1.,
        9.81*(M[0]+M[1])+t1,
        9.81*(M[1])+t1,
        t1,
        t1,
        9.81*(M[3])+t1,
        9.81*(M[4]+M[3])+t1])



for i in linspace(0,1,1e5):

    h = i
    b0 = broyden2.fdjac(bridge,x)
    xnew = new.newton(bridge,b0,x,1e-10)
    x = xnew
    print x

    if math.isnan(linalg.norm(xnew,inf)) == True:
        print i
        print x
        print 'nan break'
        break
print xnew
'''
```

Bridge 9

I changed my code a few more times before I got the hang of the morphing below is the bridge 9 morphing I used:

```
from numpy import *
import broyden2
from time import sleep
import newton_multid as new

def bridge(x):

    #h = 0.
    hhh = 0.
    s2 = (2.)**(1./2)
    t1 = 0.806047
    L = array([ 1.12 , .301, (1-h)*s2+h*.621, (1-h)*s2 + h*.261, .528, .816])
    M = array([.2226, 0.073, .0581, .0822, .128])
    #P[6] = [3.053 , .184]
    #L = array([1.12, .301,.621,.261,.528,.816])
    P = zeros((7,2))
    P[0] = [0,0]
    P[len(P)-1] = [3.053, .184]
    T = x[len(P)+3:]
    gv = array([0,-1])
    f = zeros(len(x))
    g = 9.81
    #print T

    for k in range(len(P)-2):
        P[k+1] = [x[2*k] ,x[2*k+1]]
    #print P
    for i in range(len(P)-1):
        Pd = P[i+1] - P[i]
        f[i] = linalg.norm(Pd) - L[i]

    for j in range(1,len(P)-1):
         f[i+1+2*(j-1)] = T[j-1]*(P[j-1,0]-P[j,0])/L[j-1] + T[j]*(P[j+1,0]-P[j,0])/L[j]
         f[i+2+2*(j-1)] = T[j-1]*(P[j-1,1]-P[j,1])/L[j-1] + T[j]*(P[j+1,1]-P[j,1])/L[j] - M[j-1]*9.81
         #print i+2+2*(j-1)
         #print f
    return f
```

```
#x = array([ 0.49371527,-1.32523403,1.57620047,-2.23530315,2.95255255,-2.56034921,
#  4.32614317,-2.2238246,5.3639315,-1.26309812,3.39307124, 1.54756028,
#  1.21713849,  1.21958542,  1.61421268,  2.63369606]) # with mass
#print bridge(x0)
'''
x = array([ 0.50255028, -1.32190893 , 1.6111082 , -2.20003154,  3.00064844, -2.46304852,
  4.35871246, -2.06850813,  5.37415851, -1.08419652,  3.3428752,   1.51544888,
  1.20900627,  1.23702774,  1.65440879,  2.68432643]) #with y node

x = array([ 0.40641303, -1.04366108 , 1.5007415 , -1.93945414 , 2.87437089 ,-2.27582045,
  4.2579123 , -1.98288162,  5.33134794, -1.06215586 , 3.43996945 , 1.6131379,
  1.28513755 , 1.27593052,  1.64453523 , 2.64009163]) # with L0

x = array([ 0.61334794, -0.93712555 , 0.87796872 ,-1.08057186 , 2.25950076 ,-1.38284529,
  3.67266711, -1.32843132,  4.98458163 ,-0.80034015,  4.0449679,   2.51968658,
  2.26755593,  2.21679576,  2.38788511,  3.08513343]) #With L1
'''
#x = array([ 0.86246551, -0.71453009,  1.15514561, -0.78481066,  2.56778755, -0.8514646,
#  3.97406655, -0.70186722,  5.31956946, -0.26641124,  4.81989418,  3.81710862,
#  3.71572975,  3.73254218,  3.90114034,  4.45110235]) #with L5

x = array([ 0.16478983, -1.10781059,  0.29898166, -1.3772425,   1.58297905, -1.96999094,
  2.70407743, -1.10793616,  2.89949995, -0.6174323,   3.14791254,  1.0389041,
  0.51013591,  0.58426022,  1.25139477,  2.46216238]) # with x2 and L4
#print bridge(x)
for i in linspace(0,1,1e4):

    h = i
    b0 = broyden2.fdjac(bridge,x)
    xnew = new.newton(bridge,b0,x,1e-8)
    x = xnew
    print x

    if math.isnan(linalg.norm(xnew,inf)) == True:
        print i
        print x
        print 'nan break'
        break
print xnew


#b0 = broyden2.fdjac(bridge,x)
#print new.newton(bridge,b0,x,1e-12)
```

As you can see each time I morphed a new point to the original values I got a new starting guess which I then used to morph my next point.

Broyden:

I also wrote a Broyden code that did not make it into the final cut of my program but I will include it here anyways for completeness

```
from numpy import *
import math
import time

def jacobianUpdate(bi,dx,df):
    dxt = transpose(dx)
    bdf = dot(bi,df)
    inside = dx - bdf
    num = dot(inside,dxt)
    num = dot(num,bi)
    den = dot(bi,df)
    den = dot(dxt,den)
    bip1 = bi + num/den
    return bip1

def guessUpdate(xi,bi,f):
    bif = dot(bi,f)
    xip1 = xi - bif
    return xip1
```

```
def broyden(x0,f,b,tol,lim):
    xold = x0
    i = 0
    while True:
        i += 1
        #print i
        fold = f(xold)
        xnew = guessUpdate(xold,b,fold)
        fnew = f(xnew)
        dx = transpose(array([xnew - xold]))
        df = transpose(array([fnew - fold]))
        bnew = jacobianUpdate(b,dx,df)
        #print bnew
        #bnew = fdjac(f,xnew)
        error = abs(xnew) - abs(xold)
        error = linalg.norm(error,inf)
        xold = xnew.copy()
        fold = fnew.copy()
        b = bnew.copy()
        #print xnew
        if(error < tol):
            #print 'xnew: ' , xnew
            #print 'xold: ' , xold
            #print 'i: ' ,i
            return xnew
            break
        if math.isnan(linalg.norm(xnew,inf)) == True:
            break
        if i > lim:
            print 'i: ' , i
            print 'f: ' , fnew
            break
    return xnew
```