Day 23

First, a closing question on quadrature:

How is it that the Gaussian rules are so accurate for non-polynomial functions?

Consider the 2-point (m=1) Gaussian rule.
There is a unique polynomial of degree 1 or less that interpolates the sampled values of the function f for which we want to evaluate $\int_{-1}^{1} f$.

But there is an entire 2D family of functions within P_3 that interpolate the data,
and if we choose the Gaussian sample points they all have the same integral
(because we know the 2-point Gauss rule gets the integral exactly for all p in P_3).

Thus there is an entire 2D family of interpolating cubics with the same integral,
and it is plausible that some of them approximate f quite well,
making it less surprising that we get an accurate approximation of its integral.

Illustration below.

Let's take a look at an example ...

Derive a parametrization of 2D subset of P_3 where the 2-point interpolation conditions are satisfied:

```
import sympy as sp

x1 = sp.symbols('x_1')
y0,y1 = sp.symbols('y_0,y_1')
a = sp.symbols('a_0:4')
x0 = -x1
# interpolation conditions
eq0 = sum([x0**j*a[j] for j in range(4)]) - y0
eq1 = sum([x1**j*a[j] for j in range(4)]) - y1
display(eq0)
display(eq1)
# let the family be parametrized by the quadratic and cubic coeffs a_2, a_3
sol = sp.solve((eq0,eq1),(a[0],a[1]))
display(a[0])
display(sol[a[0]])
display(a[1])
display(sol[a[1]])
```

$$a_0 - a_1 x_1 + a_2 x_1^2 - a_3 x_1^3 - y_0$$

$$a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^3 - y_1$$

$$a_0$$

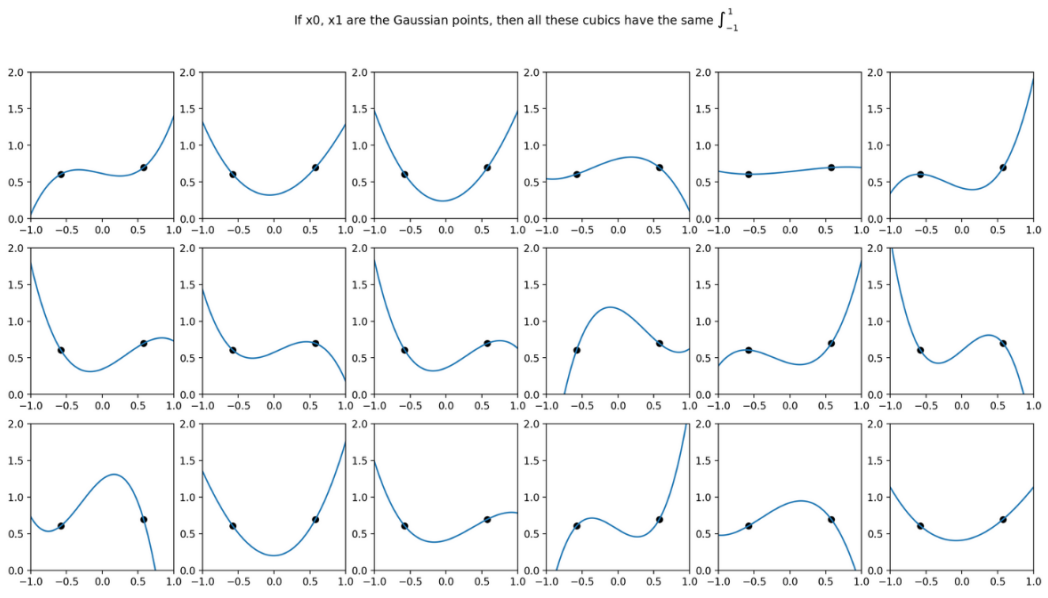
$$-a_2 x_1^2 + \frac{y_0}{2} + \frac{y_1}{2}$$

$$a_1$$

$$\frac{-2a_3 x_1^3 - y_0 + y_1}{2x_1}$$

```
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

x1 = 1/np.sqrt(3)  # Gaussian sample point (try anything else and the integral will vary)
x0 = -x1

x = np.array([x0,x1])
y = np.random.rand(len(x))


b0 = np.array([(y[0]+y[1])/2,
               (y[1]-y[0])/2/x[1],
               0,
               0])
b1 = np.array([-x[1]**2,0,1,0])
b2 = np.array([0, -x[1]**2, 0, 1])

def p(a,x):
    return np.sum([a[j]*x**j for j in range(len(a))],axis=0)

def intp(a): # exact integral over [-1,1]
    return sum([a[j]*( 1)**(j+1)/(j+1) for j in range(len(a))]) \
          -sum([a[j]*(-1)**(j+1)/(j+1) for j in range(len(a))])

xx = np.linspace(-1,1,5000)
nr,nc = 3,6
plt.figure(figsize=(nc*3,nr*3))
for k in range(nr*nc):
    plt.subplot(nr,nc,k+1)
    plt.plot(x,y,'ko')
    c = np.random.randn(2)
    a = b0 + c[0]*b1 + c[1]*b2  # coefficients of an interpolating cubic
    plt.plot(xx,p(a,xx))
    exact_integral= intp(a)
    print(exact_integral)
    plt.xlim(-1,1); plt.ylim(0,2)
plt.suptitle('If x0, x1 are the Gaussian points, then all these cubics have the same $\\int_{-1}^{1}$')
plt.savefig('temp.pdf')
```

```
1.3028194137692868
1.3028194137692866
1.3028194137692866
1.302819413769287
1.3028194137692868
1.3028194137692866
1.3028194137692863
1.3028194137692868
1.3028194137692866
1.3028194137692874
1.3028194137692868
1.3028194137692868
1.3028194137692868
1.3028194137692866
1.3028194137692868
1.3028194137692868
1.3028194137692868
1.3028194137692866
```



All these cubic (2m+1=3) functions interpolate the data,
and because the sample locations are the Gauss points
they all have the same integral,
whose value is given exactly by the Gauss rule.

Ch 8. Root-finding in multiple dimensions

a.k.a. solving systems of nonlinear equations

We seek a zero of $F : \mathbb{R}^n \to \mathbb{R}^n$

that is find $x \in \mathbb{R}^n$ such that $F(x) = 0 \in \mathbb{R}^n$

$$F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \text{choose } x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

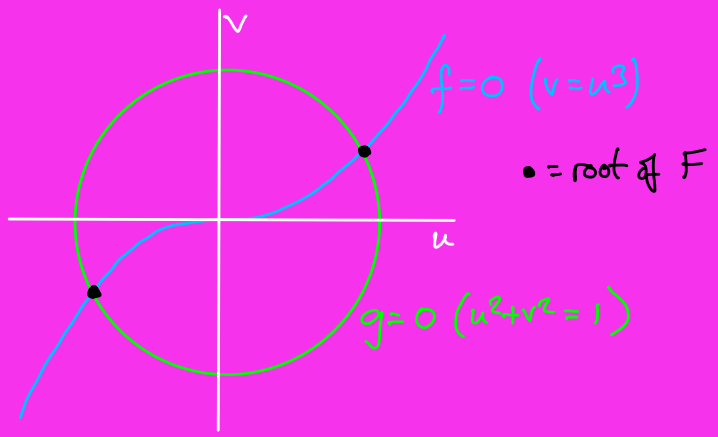
Ex ①

$n = 2$ $\quad F(x) = \begin{bmatrix} f\left(\begin{bmatrix} u \\ v \end{bmatrix}\right) \\ g\left(\begin{bmatrix} u \\ v \end{bmatrix}\right) \end{bmatrix} = \begin{bmatrix} u^3 - v \\ u^2 + v^2 - 1 \end{bmatrix}$

So we want $f\left(\begin{bmatrix} u \\ v \end{bmatrix}\right) = u^3 - v = 0$

$g\left(\begin{bmatrix} u \\ v \end{bmatrix}\right) = u^2 + v^2 - 1 = 0$

Graphically,



Our method will be iterative, starting with an initial guess $x^{(0)}$, and generating $x^{(k+1)}$ from $x^{(k)}$, k = 0,1,2,3,..., hopefully converging to a root.

Newton's method

We use exactly the same idea as in the 1-variable case (Ch 2), namely:

At $x^{(k)}$ linearize F, and use the root of the linearization as $x^{(k+1)}$.

In the n=1 case, the linearization is $L_f(x+s) = f(x) + sf'(x)$ (derivative)

Recall $f'(x) \equiv \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$.

We can view f '(x) as a number "a" such that $\lim_{h\to 0} \frac{f(x+h)-f(x)-ah}{h} = 0$.

This latter idea we can generalize to $\mathbb{R}^n$ as follows.

For $F: \mathbb{R}^n \to \mathbb{R}^n$, define its Fréchet derivative F'(x) as a (the) linear map A such that

$$\lim_{t\to 0} \frac{F(x+th)-F(x)-tAh}{t\|h\|} = 0 \quad \forall h \in \mathbb{R}^n$$ if such a map exists.

In particular, the limit must be zero for the choice $h = e_j$, the jth standard basis vector for $\mathbb{R}^n$

For this choice of h, the ith component of the LHS above is

$$\lim_{t\to 0} \frac{f_i(x+te_j) - f_i(x) - t a_{ij}}{t} \quad \text{where } a_{ij} = (A)_{ij}$$

This is $\left.\frac{\partial f_i}{\partial x_j}\right|_x$

Thus if the Frechet derivative exists, the i,j element of its matrix is

$$a_{ij} = \frac{\partial f_i}{\partial x_j}$$

Thus when the Frechet derivative exists, its matrix is ...

$$F'(x) = \begin{bmatrix} \partial_1 f_1 & \partial_2 f_1 & \cdots & \partial_n f_1 \\ \partial_1 f_2 & \partial_2 f_2 & \cdots & \partial_n f_2 \\ \vdots & & & \\ \partial_1 f_n & \partial_2 f_n & \cdots & \partial_n f_n \end{bmatrix},$$ the "jacobian matrix" of partial derivatives.

which we can obtain by one or more of the 3 differentiation methods we discussed recently.

Note: having partial derivatives at a point does NOT imply the Frechet derivative exists there

Example: $f(x, y) = \frac{xy}{x^2 + y^2}$

- **Partial derivatives at** $(0, 0)$**:**
  - The partial derivative with respect to x, $f_x(0, 0)$, is the limit of $\frac{f(h, 0) - f(0, 0)}{h}$ as $h \to 0$. Since $f(h, 0) = \frac{h \cdot 0}{h^2 + 0^2} = 0$, the limit is 0.
  - Similarly, the partial derivative with respect to y, $f_y(0, 0)$, is also 0.
- **Non-differentiability at** $(0, 0)$**:**
  - The function is not even continuous at $(0, 0)$ because the limit as (x, y) approaches $(0, 0)$ depends on the path taken.
  - Along the line $y = x$, the function becomes $\frac{x^2}{x^2 + x^2} = \frac{x^2}{2x^2} = \frac{1}{2}$ for $x \neq 0$.
  - Along the line $y = 2x$, the function becomes $\frac{x(2x)}{x^2 + (2x)^2} = \frac{2x^2}{5x^2} = \frac{2}{5}$ for $x \neq 0$.
  - Since the function approaches different values from different paths, the limit at $(0, 0)$ does not exist, so the function is not continuous and therefore not differentiable at the origin.

But if partial derivatives exist and are continuous then the F-derivative does exist (and its matrix is the jacobian).

Newton's method is to approximate F near x by

$$L_F(x+s) \equiv F(x) + F'(x)\, s \overset{\text{set}}{=} 0$$

Thus we solve the linear system

$$F'(x)\, s = -F(x) \quad \text{for } s$$

or more specifically,

$$F'(x^{(k)})\, s^{(k)} = -F(x^{(k)}) \quad \text{for } s^{(k)} \text{ the } k^{\text{th}} \text{ Newton step}$$

and set $x^{(k+1)} = x^{(k)} + s^{(k)}$.

In example (1) $F(x) = \begin{bmatrix} f(\begin{bmatrix} u \\ v \end{bmatrix}) \\ g(\begin{bmatrix} u \\ v \end{bmatrix}) \end{bmatrix} = \begin{bmatrix} u^3 - v \\ u^2 + v^2 - 1 \end{bmatrix}$,

$$F'(x) = \begin{bmatrix} \partial_u f & \partial_v f \\ \partial_u g & \partial_v g \end{bmatrix} = \begin{bmatrix} 3u^2 & -1 \\ 2u & 2v \end{bmatrix}$$

If our initial guess were $x^{(0)} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, then $F(x^{(0)}) = \begin{bmatrix} 2^3 - 1 \\ 2^2 + 1^2 - 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \end{bmatrix}$,

$F'(x^{(0)}) = \begin{bmatrix} 3 \cdot 2^2 & -1 \\ 2 \cdot 2 & 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 12 & -1 \\ 4 & 2 \end{bmatrix}$ and $x^{(1)} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} + s$ where $\begin{bmatrix} 12 & -1 \\ 4 & 2 \end{bmatrix} s = -\begin{bmatrix} 7 \\ 4 \end{bmatrix}$.

Let's implement it in code and see what happens ...

[Note to self: try x0 = [.1,.1] among others.]

Implementation of multi-D Newton's method

```
import numpy as np

def newton(F,Fprime,x0,tol):

def myF(x):

def myFprime(x):

x =      # initial guess at root
z = newton(myF, myFprime, x, 1.e-12 )
```

Implementation of multi-D Newton's method

```
import numpy as np
```

```
def newton(F,Fprime,x0,tol):
    x = np.array(x0,dtype=float)
    k = 0
    while True:
        Fval = F(x)
        Fprimeval = Fprime(x)
        s = np.linalg.solve(Fprimeval,-Fval)
        x += s
        k += 1
        if np.linalg.norm(s) <= tol:
            print(f'{k} iterations used')
            return x

def myF(x):
    u,v = x
    return np.array([u**3-v, u**2+v**2-1])

def myFprime(x):
    u,v = x
    return np.array([[3*u**2, -1],
                     [2*u,   2*v]])

x = [2,1]     # initial guess at root
tol = 1e-12

z = newton(myF, myFprime, x, tol )
print('Approximate root is',z)
print('Residual is',myF(z)) # sanity check: should be very small
```

```
7 iterations used
Approximate root is [0.82603136 0.56362416]
Residual is [-1.11022302e-16  0.00000000e+00]
```

Accurate to machine precision in 7 iterations.

Initial guesses given by Yingkun, Tien, me, and Joel ...

```
x = [1,0]     # initial guess at root
tol = 1e-12

z = newton(myF, myFprime, x, tol )
print('Approximate root is',z)
print('Residual is',myF(z)) # sanity check: should be very small
```

```
7 iterations used
Approximate root is [0.82603136 0.56362416]
Residual is [1.11022302e-16 0.00000000e+00]
```

Initial guess closer to root (in 2-norm),
but 7 iterations still required.

```
x = [1,0.5]     # initial guess at root
tol = 1e-12

z = newton(myF, myFprime, x, tol )
print('Approximate root is',z)
print('Residual is',myF(z)) # sanity check: should be very small
```

```
5 iterations used
Approximate root is [0.82603136 0.56362416]
Residual is [ 0.00000000e+00 -2.22044605e-16]
```

Even closer initial guess,
only 5 iterations required.

```
c = np.cos(7/6*np.pi)
s = np.sin(7/6*np.pi)
x = [c,s]     # initial guess at root
tol = 1e-12

z = newton(myF, myFprime, x, tol )
print('Approximate root is',z)
print('Residual is',myF(z)) # sanity check: should be very small
```

```
5 iterations used
Approximate root is [-0.82603136 -0.56362416]
Residual is [-1.11022302e-16  0.00000000e+00]
```

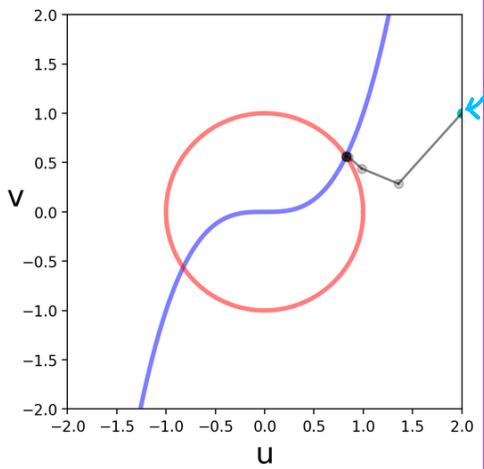
A yet closer initial guess,
this time from Joel for the 3rd quadrant root,
requires 5 iterations for the desired accuracy.

Below, I've written a more elaborate version that makes a picture of what's happening.

```
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt

def newton(F,Fprime,x,tol):
    x = np.array(x,dtype=float)  # floating point copy of x in case it comes in as ints
    while True:
        Fval = F(x)
        Fpval = Fprime(x)
        s = np.linalg.solve(Fpval,-Fval)  # solve for our Newton step
        newx = x + s
        plt.plot([x[0],newx[0]],[x[1],newx[1]],'k',alpha=0.5)
        plt.plot(newx[0],newx[1],'ko',alpha=0.2)
        x = newx
        if np.linalg.norm(s) < tol:
            return x  # done!

def myF(x):
    u,v = x
    return np.array([ u**3 -v, u**2 + v**2 - 1 ])

def myFprime(x):
    u,v = x
    return np.array([[ 3*u**2, -1 ],
                     [ 2*u,    2*v]])

def doit(x0,cx,cy,r):
    plt.subplot(111,aspect=1)

    # draw the curves
    u = np.linspace(-1.5,1.5,400)
    v = u**3
    plt.plot(u,v,'b',lw=3,alpha=0.5)
    t = np.linspace(0,2*np.pi,400)
    x,y = np.cos(t),np.sin(t)
    plt.plot(x,y,'r',lw=3,alpha=0.5)

    x = x0 #[1/2,1/2]
    plt.plot(x[0],x[1],'co')
    z = newton(myF,myFprime,x,1e-12)
    print('Approx root',z)
    # what's the residual at our approx root z?
    print('residual at z:',myF(z))
    #r = 1.5
    plt.xlim(cx-r,cx+r)
    plt.ylim(cy-r,cy+r)
    plt.xlabel('u',fontsize=20)
    plt.ylabel('v',fontsize=20,rotation=0)
    plt.savefig('temp.pdf');  # better render
    return z
```

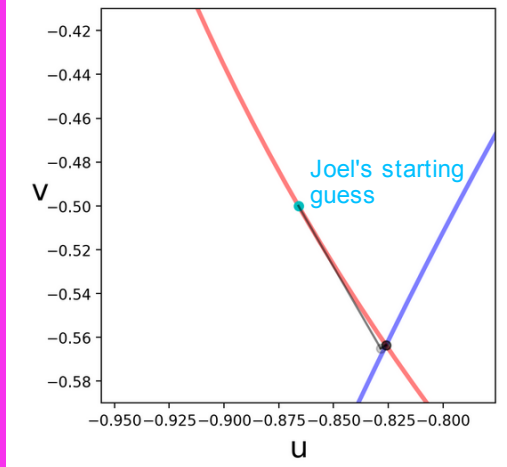
```
doit([2,1],0,0,2)
```

```
Approx root [0.82603136 0.56362416]
residual at z: [-1.11022302e-16  0.00000000e+00]

array([0.82603136, 0.56362416])
```




```
doit([c,s],c,s,.09)
```

```
Approx root [-0.82603136 -0.56362416]
residual at z: [-1.11022302e-16  0.00000000e+00]

array([-0.82603136, -0.56362416])
```

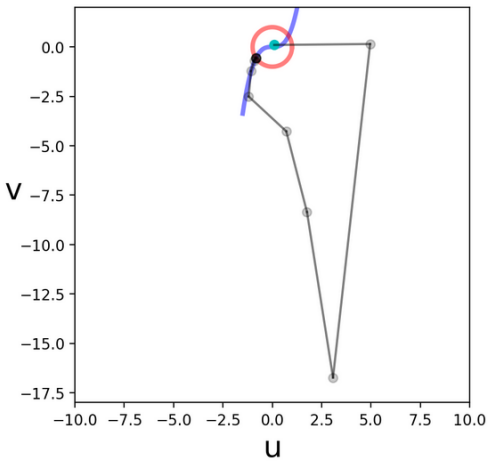


An initial guess close to the origin (where, we oberved, the jacobian is singular).

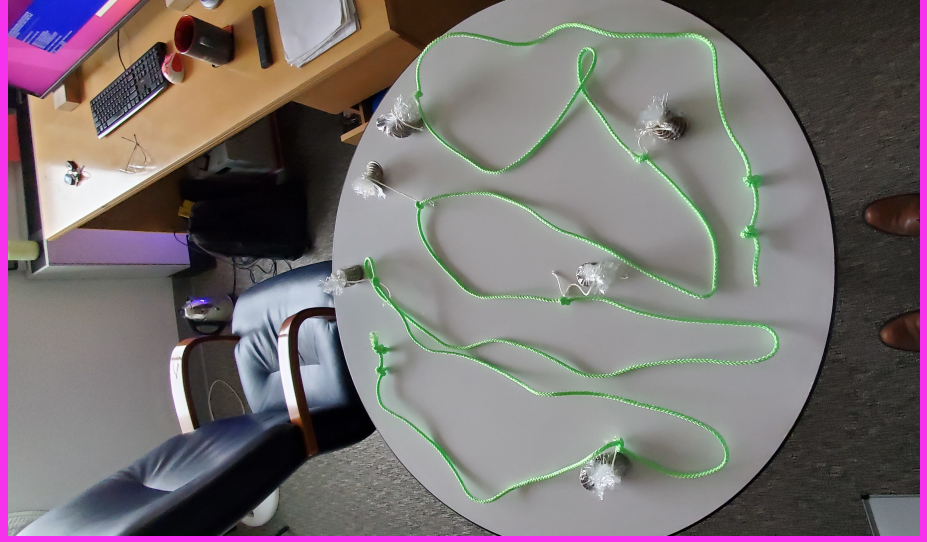
Newton jumps far from the root before coming back and converging. Guessing it took about 12 iterations in this case.

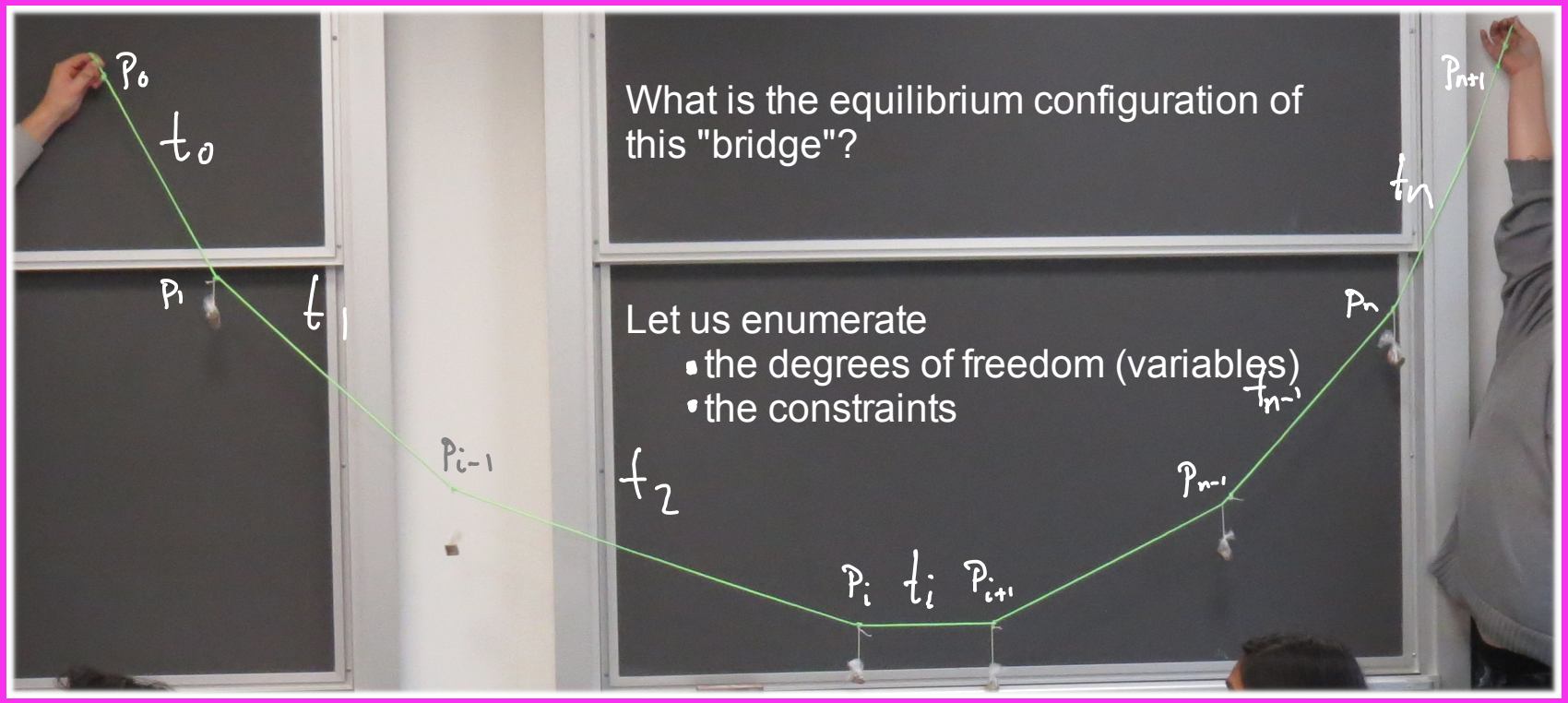
```
doit([.1,.1],0,-8,10)
```

```
Approx root [-0.82603136 -0.56362416]
residual at z: [-1.11022302e-16  0.00000000e+00]

array([-0.82603136, -0.56362416])
```



Example 2 : a toy suspension bridge





Variables : $p_1, p_2, \ldots, p_n$ each in $\mathbb{R}^2$: $2n$ scalars

$t_0, t_1, \ldots t_n$: $n+1$ scalar

$3n+1$ scalar unknowns

Constraints: (i) mechanical equilibrium at each of the n knots.

(ii) length of string segments fixed

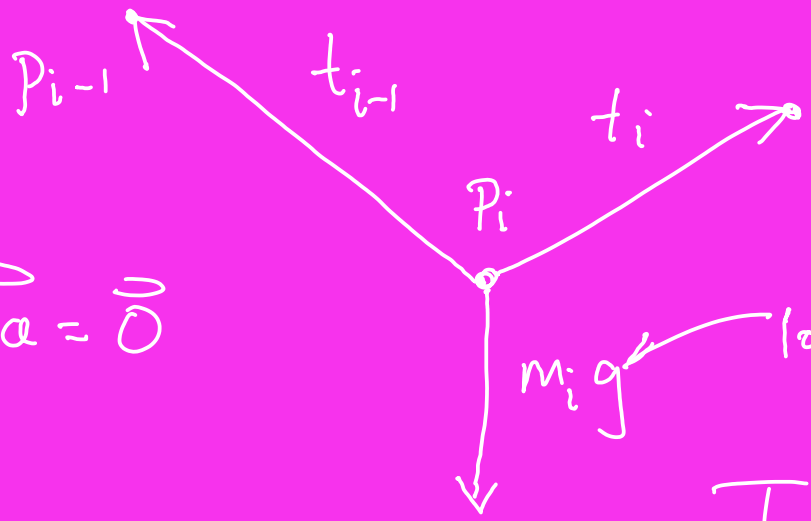
(ii) length of segment from p_i to p_{i+1}

$\| p_{i+1} - p_i \|_2 = l_i$ where l_i is a given constant.

$i = 0, 1, \ldots, n$.

$n+1$

(i)



$i = 1, \ldots, n$.

net force $= \vec{0}$

local grav. accel.

n vector eqns

$= 2n$ scalar equations

Total scalar constraints $= 3n+1$ ☺

Thus we have 3n+1 nonlinear equations in 3n+1 variables
- a problem that should be solvable with the multidimensional Newton's method.