Reality Check 1: Kinematics of the Stewart Platform

Riley Auman, Bryant Arias, Pieter Alley, Addison Armistead, Samantha Bennett

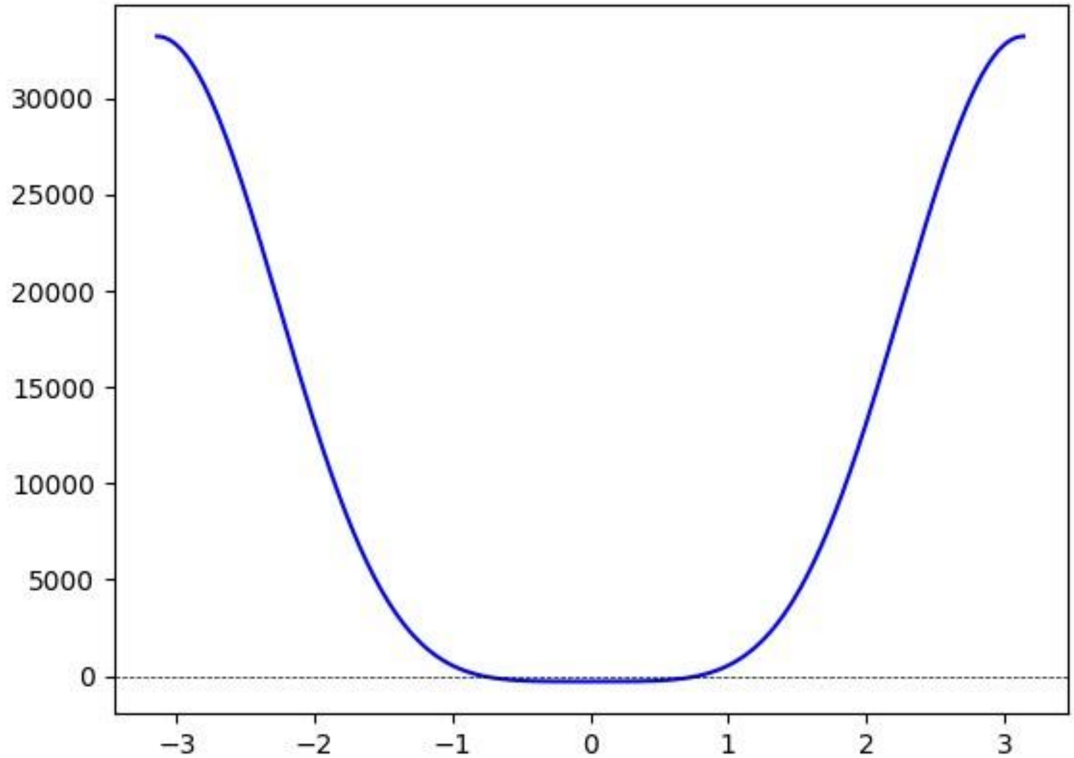
[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
import scipy
import math
```

1. **Write a Matlab function file for $f(\theta)$. The parameters $L_1, L_2, L_3, \gamma, x_1, x_2, y_2$ are fixed constants, and the strut lengths p_1, p_2, p_3 will be known for a given pose. To test your code, set the parameters $L_1 = 2, L_2 = L_3 = \sqrt{2}, \gamma = \pi/2, p_1 = p_2 = p_3 = \sqrt{5}$ from Figure 1.15. Then, substituting $\theta = -\pi/4$ or $\theta = \pi/4$, corresponding to Figures 1.15(a, b), respectively, should make $f(\theta) = 0$.**

[2]:
```python
def f(theta):
    A2 = L3 * math.cos(theta) - x1
    A3 = L2 * (math.cos(theta)*math.cos(gamma) - math.sin(theta)*math.
↪sin(gamma)) - x2
    B2 = L3 * math.sin(theta)
    B3 = L2 * (math.cos(theta)*math.sin(gamma) + math.sin(theta)*math.
↪cos(gamma)) - y2

    N1 = B3*(p2**2 - p1**2 - A2**2 - B2**2) - B2*(p3**2 - p1**2 - A3**2 - B3**2)
    N2 = -A3*(p2**2 - p1**2 - A2**2 - B2**2) + A2*(p3**2 - p1**2 - A3**2 -␣
↪B3**2)
    D = 2*(A2*B3 - B2*A3)

    out = N1**2 + N2**2 - p1**2 * D**2

    return out

L1, L2, L3 = 2, math.sqrt(2), math.sqrt(2)
p1, p2, p3 = math.sqrt(5), math.sqrt(5), math.sqrt(5)
x1, x2, y2 = 4, 0, 4
gamma = math.pi/2
```

2. **Plot $f(\theta)$ on $[-\pi, \pi]$. As a check of your work, there should be roots at $\pm\pi/4$.**

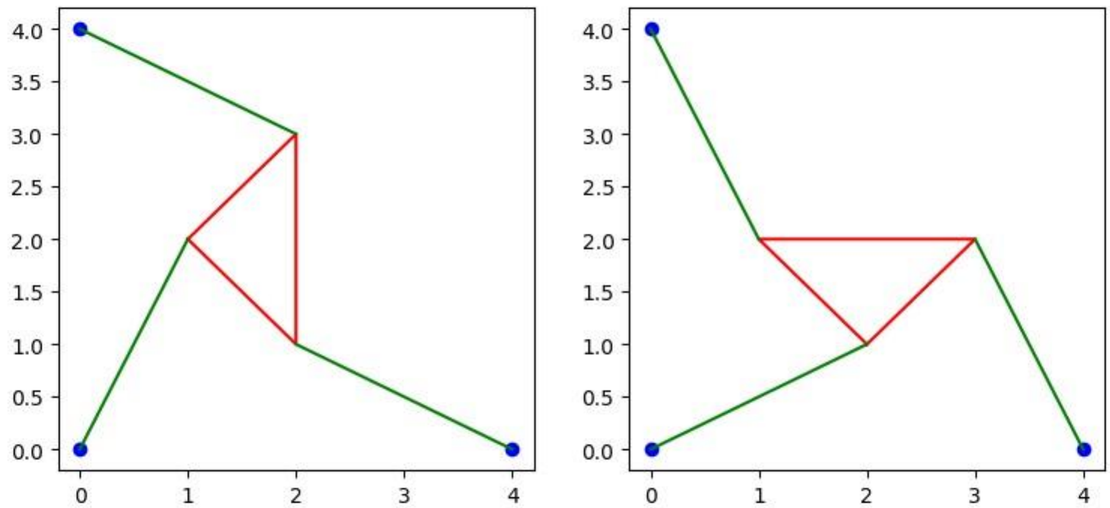
[3]:
```python
X = np.linspace(-math.pi, math.pi, 500)
Y = [f(x) for x in X]
plt.plot(X, Y, 'b-')
```



3. Reproduce Figure 1.15. In addition, draw the struts.

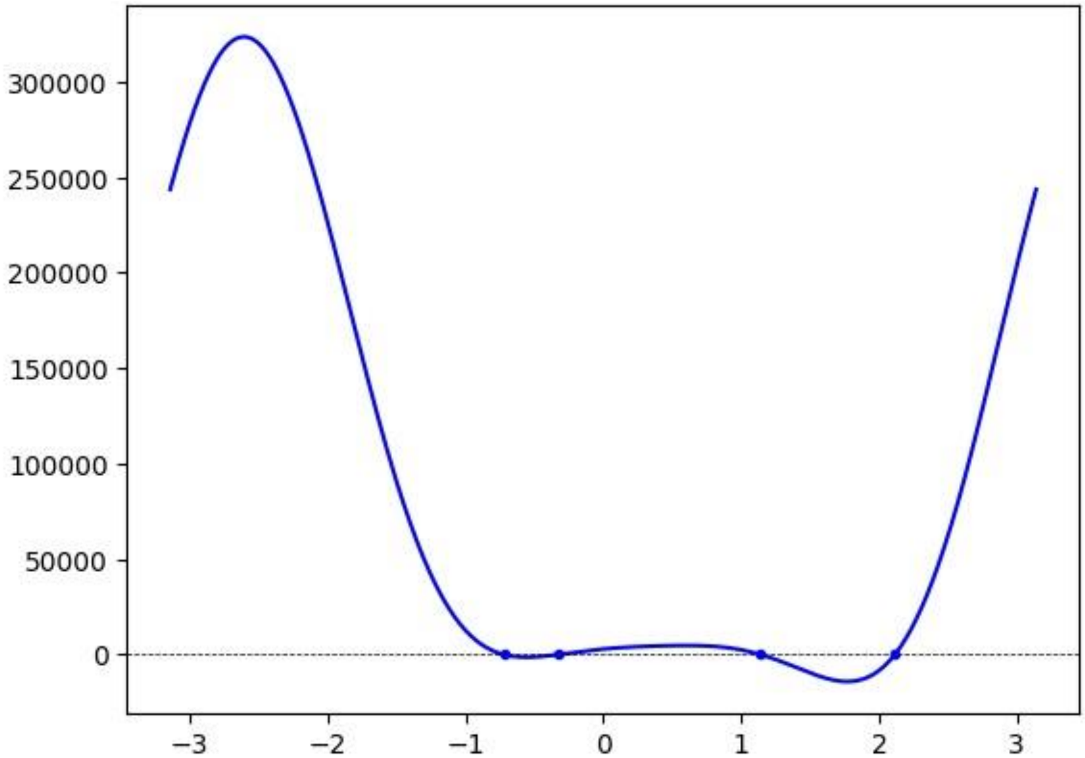
[4]:
```
def spPlot(ax, u, v, p):
    ax.plot([u[0],  [1],  [2],  [0]],[v[0],  [1],  [2],  [0]], 'r')
    ax.plot([0, x1, x2],[0, 0, y2], 'bo')
    ax.plot(p[0][0],  [0][1], 'g')
    ax.plot(p[1][0],  [1][1], 'g')
    ax.plot(p[2][0],  [2][1], 'g')

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(9, 4))

u = [1, 2, 2]
v = [2, 1, 3]
p = [[[0, 1], [0, 2]], [[2, 4], [1, 0]], [[0, 2], [4, 3]]]
spPlot(ax1,u,v,p)

u = [2, 3, 1]
```

```
v = [1, 2, 2]
p = [[[0, 2], [0, 1]], [[3, 4], [2, 0]], [[0, 1], [4, 2]]]
spPlot(ax2,u,v,p)
```



4. **Solve the forward kinematics problem specified by** $x_1 = 5, (x_2, y_2) = (0, 6), L_1 = L_3 = 3, L_2 = 3\sqrt{2}, \gamma = \pi/4, p_1 = p_2 = 5, p_3 = 3$**. Begin by plotting** $f(\theta)$**. Use an equation solver to find all four poses, and plot them. Verify that** p_1, p_2**, and** p_3 **are the lengths of the struts in your plot.**

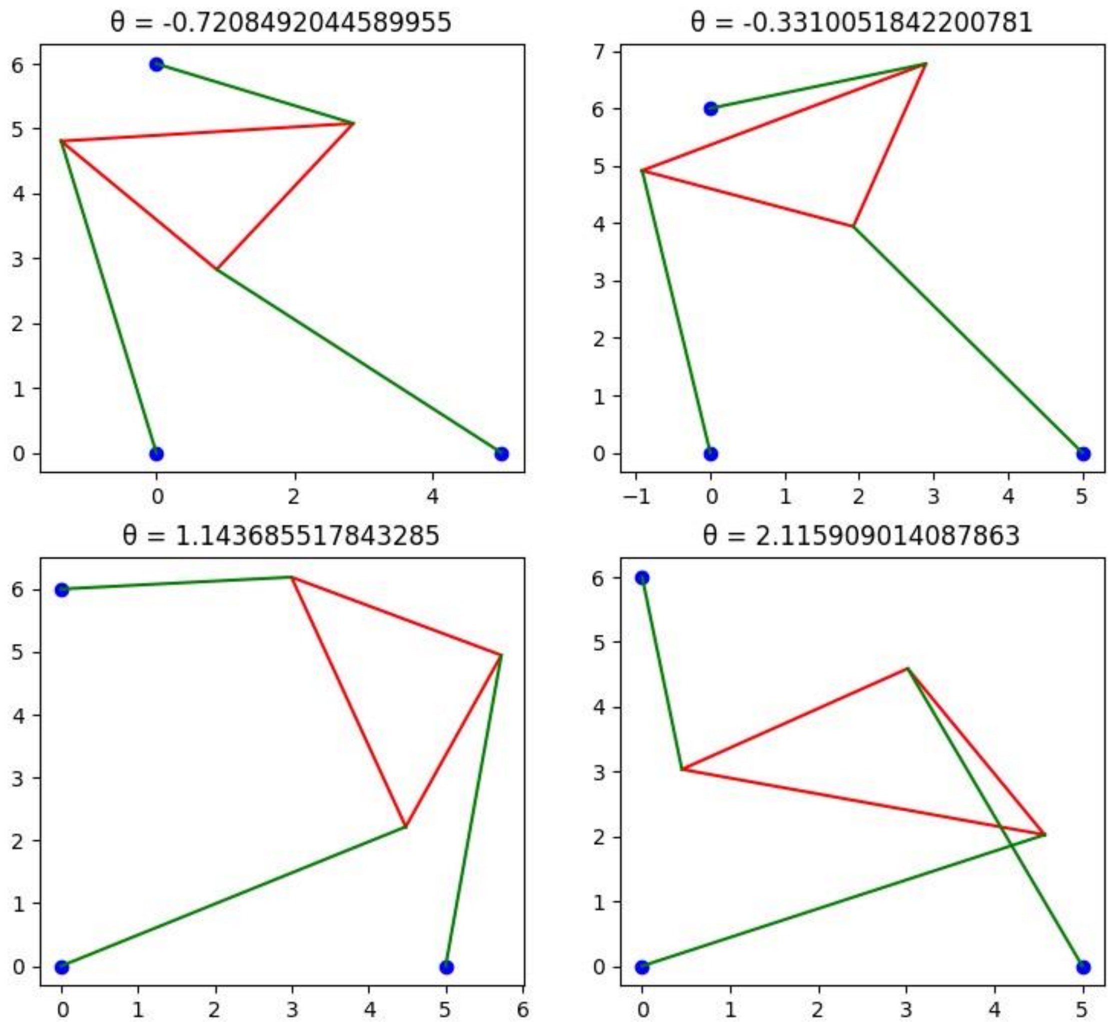
[5]:
```
L1, L2, L3 = 3, 3*math.sqrt(2), 3
p1, p2, p3 = 5, 5, 3
x1, x2, y2 = 5, 0, 6
gamma = math.pi/4

X = np.linspace(-math.pi, math.pi, 500)
Y = [f(x) for x in X]
plt.plot(X, Y, 'b-')
plt.axhline(0, color='black', lw = 0.5, ls='--')

vf = np.vectorize(f)
roots = scipy.optimize.fsolve(vf, [-1, 0, 1, 2])
for root in roots:
    plt.plot(root, 0, 'b.')
```



[6]:
```
def platformPoints(theta):
    A2 = L3 * math.cos(theta) - x1
    A3 = L2 * (math.cos(theta)*math.cos(gamma) - math.sin(theta)*math.
 ↪sin(gamma)) - x2
    B2 = L3 * math.sin(theta)
    B3 = L2 * (math.cos(theta)*math.sin(gamma) + math.sin(theta)*math.
 ↪cos(gamma)) - y2

    N1 = B3*(p2**2 - p1**2 - A2**2 - B2**2) - B2*(p3**2 - p1**2 - A3**2 - B3**2)
    N2 = -A3*(p2**2 - p1**2 - A2**2 - B2**2) + A2*(p3**2 - p1**2 - A3**2 -␣
 ↪B3**2)
    D = 2*(A2*B3 - B2*A3)

    x = N1 / D
    y = N2 / D
    u = [x,(x + L2*math.cos(theta + gamma)),(x + L3*math.cos(theta))]
    v = [y,(y + L2*math.sin(theta + gamma)),(y + L3*math.sin(theta))]
    p = [[[0,x],[0,y]], [[x1,u[2]],[0,v[2]]],[[x2,u[1]],[y2,v[1]]]]

    return u, v, p
```

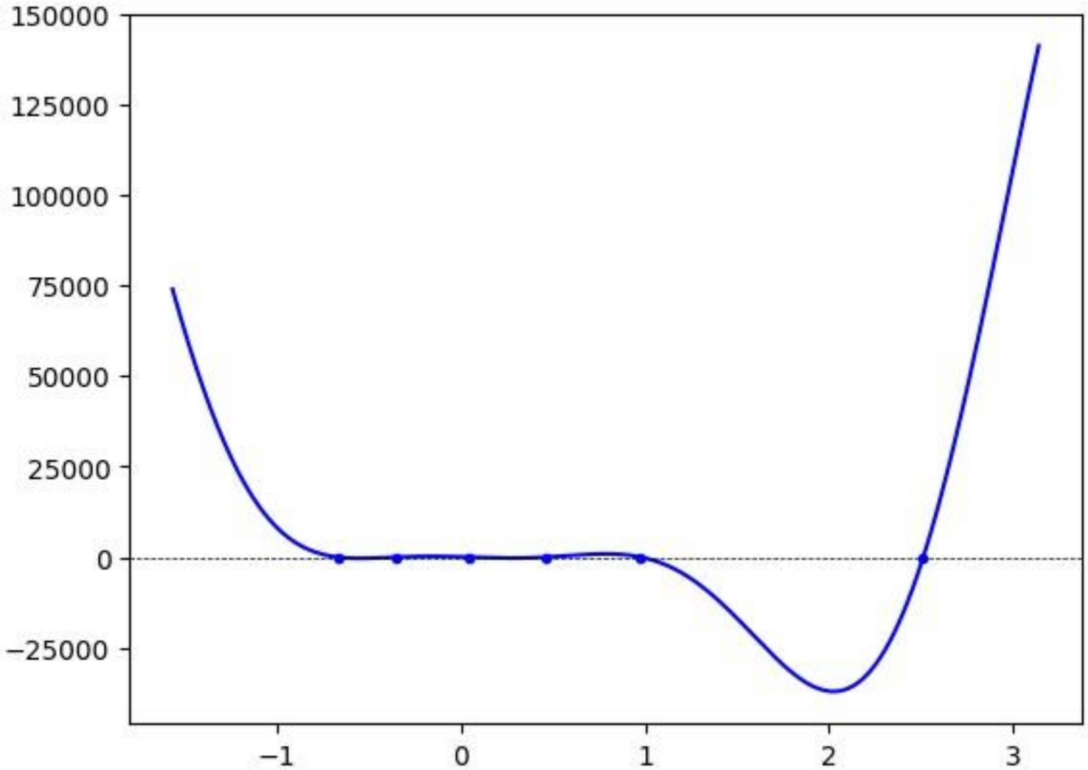
```
fig, axs = plt.subplots(2, 2, figsize=(9, 8))

for i in range(2):
    for j in range(2):
        u, v, p = platformPoints(roots[(i*2)+j])
        axs[i, j].set_title(f'  = {roots[(i*2)+j]}')
        spPlot(axs[i][j],u,v,p)
        print(f'Plot {i*2+j+1}')
        print(f'   p1:{math.dist([p[0][0][0],p[0][1][0]], [p[0][0][1],␣
 ↪p[0][1][1]]):.5f},',
              f' p2:{math.dist([p[1][0][0],p[1][1][0]], [p[1][0][1], p[1][1][1]]):
 ↪.5f},',
              f' p3:{math.dist([p[2][0][0],p[2][1][0]], [p[2][0][1], p[2][1][1]]):
 ↪.5f}')
```

```
Plot 1
   p1:5.00000,  p2:5.00000,  p3:3.00000
Plot 2 p1:5.00000, p2:5.00000,
   p3:3.00000
Plot 3 p1:5.00000, p2:5.00000,
   p3:3.00000
Plot 4 p1:5.00000, p2:5.00000,
   p3:3.00000
```

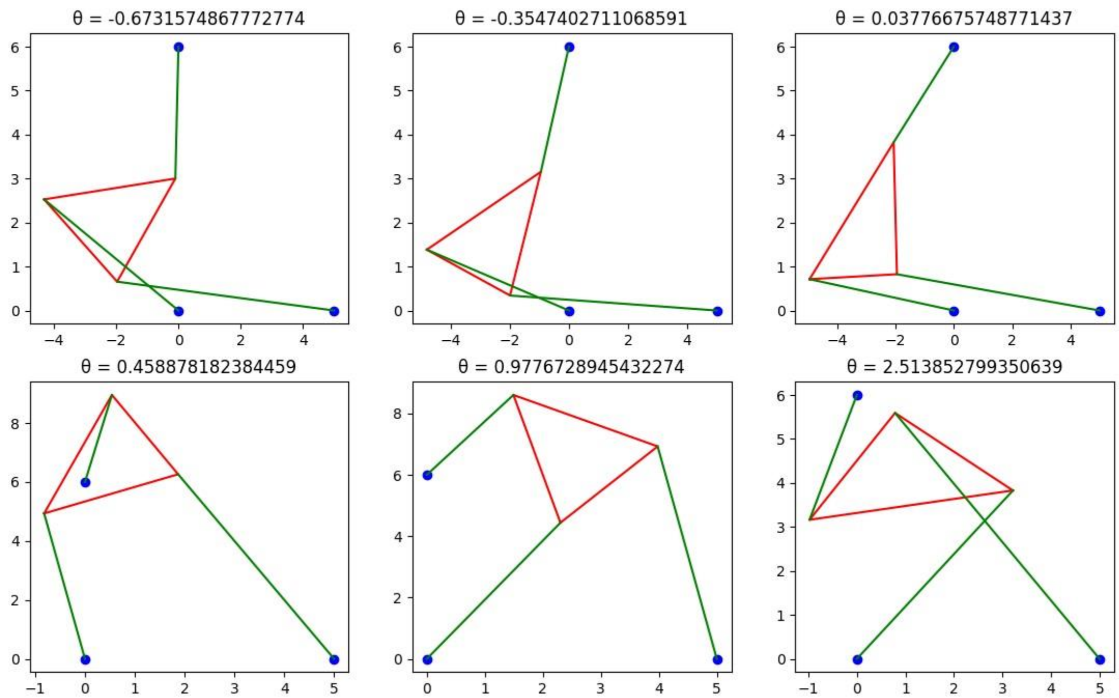


5. Change strut length to $p_2 = 7$ and re-solve the problem. For these parameters, there are six poses.

[7]:
```
p2 = 7

X = np.linspace(-math.pi/2, math.pi, 500)
Y = [f(x) for x in X]
plt.plot(X, Y, 'b-')
plt.axhline(0, color='black', lw = 0.5, ls='--')

vf = np.vectorize(f)
roots = scipy.optimize.fsolve(vf, [-0.7, -0.3, 0, 0.5, 1, 2.5])
for root in roots:
    plt.plot(root, 0, 'b.')
```

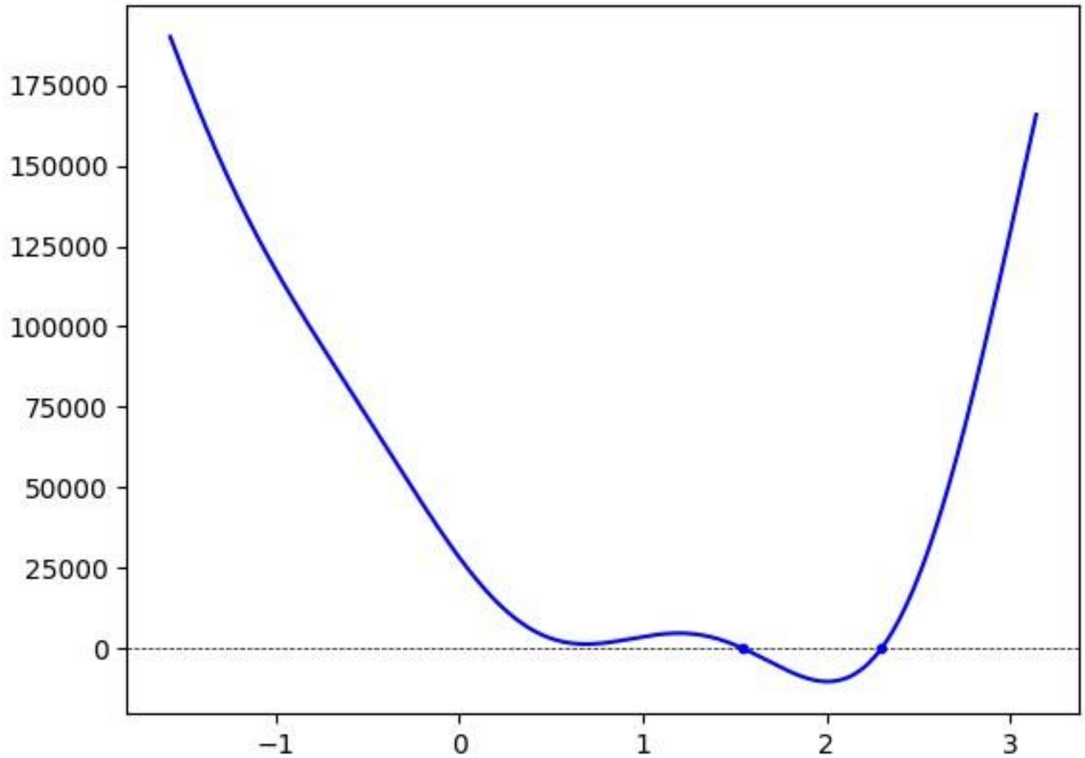



```
[8]: fig, axs = plt.subplots(2, 3, figsize=(13.5, 8))

     for i in range(2):
         for j in range(3):
             u, v, p = platformPoints(roots[(i*3)+j])
             axs[i, j].set_title(f'=   {roots[(i*3)+j]}')
             spPlot(axs[i][j],u,v,p)
```



6. **Find a strut length p_2, with the rest of the parameters as in Step 4, for which there are only two poses.**

[9]:
```
p2 = 9

X = np.linspace(-math.pi/2, math.pi, 500)
Y = [f(x) for x in X]
plt.plot(X, Y, 'b-')
plt.axhline(0, color='black', lw = 0.5, ls='--')

vf = np.vectorize(f)
roots = scipy.optimize.fsolve(vf, [1.6, 2.2])
for root in roots:
    plt.plot(root, 0, 'b.')
```



7. **Calculate the intervals in p_2, with the rest of the parameters as in Step 4, for which there are 0,2,4, and 6 poses, respectively.**

[10]:
```
def step7(p2_res, theta_res):
    prev_out = f(-math.pi)
    prev_poseCount = 0
    poseCount = 0
    prev_p2 = 0

    for i in range(p2_res):
        global p2
        p2 = 10 * i / p2_res
        for j in range(theta_res):
            out = f(((2 * math.pi) * j / theta_res) - math.pi)
            if (prev_out * out) < 0:
                poseCount = poseCount + 1
            prev_out = out
        if poseCount != prev_poseCount:
            print(f'{prev_poseCount} poses: ({prev_p2}, {p2})')
            prev_p2 = p2
        prev_poseCount = poseCount
        poseCount = 0
```

```
    print(f'{prev_poseCount} poses: ({prev_p2}, {p2})')

step7(1000, 360)
```

```
0 poses: (0, 3.72)
2 poses: (3.72, 4.87)
4 poses: (4.87, 6.97)
6 poses: (6.97, 7.03)
4 poses: (7.03, 7.85)
2 poses: (7.85, 9.27)
0 poses: (9.27, 9.99)
```

8. Derive or look up the equations representing the forward kinematics of the three-dimensional, six-degrees-of-freedom Stewart platform. Write a program and demonstrate its use to solve the forward kinematics.

[11]:
```
# Rotation helpers
def rotx(alpha):
    return np.array([
        [1, 0, 0],
        [0, np.cos(alpha), -np.sin(alpha)],
        [0, np.sin(alpha),  np.cos(alpha)]
    ])

def roty(beta):
    return np.array([
        [np.cos(beta), 0, np.sin(beta)],
        [0, 1, 0],
        [-np.sin(beta), 0, np.cos(beta)]
    ])

def rotz(gamma):
    return np.array([
        [np.cos(gamma), -np.sin(gamma), 0],
        [np.sin(gamma),  np.cos(gamma), 0],
        [0, 0, 1]
    ])

def fk_eqs(x, B, P, L):
    phi, theta, psi = x[:3]
    tvec = x[3:]
    R = rotz(psi) @ roty(theta) @ rotx(phi)
    F = np.zeros(6)
    for i in range(6):
        Xi = R @ P[:, i] + tvec
        F[i] = np.linalg.norm(Xi - B[:, i]) - L[i]
    return F
```

```python
# Finite difference Jacobian with damping
def jacobian_fd(func, x, args=(), eps=1e-6):
    n = len(x)
    J = np.zeros((6, n))
    fx = func(x, *args)
    for j in range(n):
        dx = np.zeros(n)
        dx[j] = eps
        fx_eps = func(x + dx, *args)
        J[:, j] = (fx_eps - fx) / eps
    return J

def newton_damped(func, x0, args=(), tol=1e-6, max_iter=50, lam=1e-3):
    x = x0.copy()
    for i in range(max_iter):
        F = func(x, *args)
        normF = np.linalg.norm(F)
        print(f"Iter {i}, Residual = {normF:.3e}")
        if normF < tol:
            return x
        J = jacobian_fd(func, x, args)
        try:
            dx = np.linalg.solve(J.T @ J + lam*np.eye(len(x)), -J.T @ F)
        except np.linalg.LinAlgError:
            print("Jacobian is singular.")
            break
        x += dx
    print("Failed to converge.")
    return x

# 1. Geometry
rb = 100
rp = 50
beta = np.arange(6) * 60 * np.pi / 180
B = np.vstack((rb * np.cos(beta), rb * np.sin(beta), np.zeros(6)))
alpha = beta + 30 * np.pi / 180
P = np.vstack((rp * np.cos(alpha), rp * np.sin(alpha), np.zeros(6)))

# 2. Known pose
phi = 10 * np.pi / 180
theta = 5 * np.pi / 180
psi = 15 * np.pi / 180
t = np.array([20, 30, 40])
R = rotz(psi) @ roty(theta) @ rotx(phi)
L = np.zeros(6)
for i in range(6):
```

```python
    Xi = R @ P[:, i] + t
    L[i] = np.linalg.norm(Xi - B[:, i])

# 3. Solve forward kinematics
x0 = np.zeros(6)
x0[3:] = np.array([10, 10, 10])  # Better guess for translation
sol = newton_damped(fk_eqs, x0, args=(B, P, L))

# 4. Output
phi_sol, theta_sol, psi_sol = sol[:3]
t_sol = sol[3:]
print("\nRecovered pose (radians and units):")
print(f" phi   = {phi_sol:.6f}, theta = {theta_sol:.6f}, psi = {psi_sol:.6f}")
print(f" t     =  {t_sol[0]:.6f}, {t_sol[1]:.6f}, {t_sol[2]:.6f}]")
```

```
Iter 0, Residual = 6.728e+01
Iter 1, Residual = 3.569e+01
Iter 2, Residual = 8.894e+00
Iter 3, Residual = 8.179e-01
Iter 4, Residual = 9.071e-02
Iter 5, Residual = 1.892e-03
Iter 6, Residual = 2.077e-06
Iter 7, Residual = 1.642e-09

Recovered pose (radians and units):
phi        = -0.184498
theta      = -0.063291
psi        = 0.528181
t          = [14.101778, 20.107716, 14.247459]
```