2025

Project 1. Forward and inverse kinematics

AIM 5003 NUMERICAL METHODS

GREGORY E. SCHWARTZ | 800777831

Abstract

This project explores the numerical solution of a two-link robotic manipulator's forward and inverse kinematics using three root-finding methods: the Bisection Method, Fixed-Point Iteration, and Newton's Method. Forward kinematics is used to compute the (x,y) position of the manipulator's end-effector given joint angles θ1 and θ2, while inverse kinematics determines the required angles for a desired end-effector position.

Introduction

(1) Develop a numerical function to compute the forward kinematics of a manipulator (from equations (1) and (2)) given the angles θ1 and θ2.

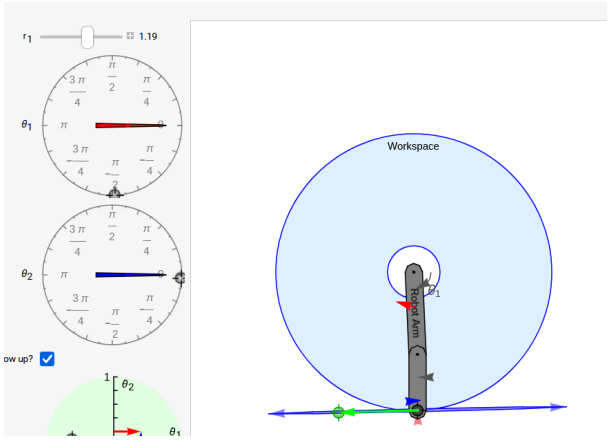
a. L1 = link 1's fixed length
b. L2 = link 2's fixed length
c. End – Effector is located at the end of link 2. The robot's claw.
d. θ1 = Angle of the first link (in radians).
e. θ2 = Angle of the second link (in radians).
f. (x,y) = The computed position of the robot hand (end-effector).

$$x = L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2)$$

$$y = L_1 \sin(\theta_1) + L_2 \sin(\theta_1 + \theta_2)$$

(2) Describes the set of x–y points which are which are within the reach of the manipulator.

a. The reach of the manipulator is defined as the result of how far the robot hand can extend from the base and how close it can get to the base.
b. Maximum Reach = L1+L2 ; outermost boundary
c. Minimum Reach = | L1 – L2 | (if non zero) ; innermost boundary
d. The set of (x, y) Points that are within the reach of the manipulator is given by:
    i. $|L1 - L2| \leq r \leq L1 + L2, where\ r = \sqrt{(x^2 + y^2)}$



(3) Develop a numerical function to compute the inverse kinematic solution of the manipulator (i.e., determine θ1 and θ2 in equations (1) and (2)) given the hand location x and y.)

$x = L1\cos(\theta_1) + L2\cos(\theta_1 + \theta_2)$ (1)

$y = L1\sin(\theta_1) + L2\sin(\theta_1 + \theta_2)$ (2)

Next Isolate

$$x - L1\cos(\theta_1) = L2\cos(\theta_1 + \theta_2)$$
$$y - L1\sin(\theta_1) = L2\sin(\theta_1 + \theta_2)$$

Next Combine

$$(x - L1\cos(\theta_1))^2 = L2^2\ \cos^2(\theta_1 + \theta_2)^2$$

$$(y - L1\sin(\theta_1))^2 = L2^2\sin(\theta_1 + \theta_2)^2$$

Given: $\cos(\theta_1 + \theta_2)^2 + \sin(\theta_1 + \theta_2)^2 = 1$

Therefore:

$$(x - L1\cos(\theta_1))^2 + (y - L1\sin(\theta_1))^2 = L2^2$$

This equation involves only one unknown θ_1 when given (x,y,L1,L2)

Solve this equation for θ_1(numerically) and plug the value of $\theta_1$in the original equations and solve for θ_2 Numerically.

In order to do this we can create the following equation:

$$f(\theta_1) = (x - L1\cos(\theta_1))^2 + (y - L1\sin(\theta_1))^2 - L2^2$$

And search for a θ_1 where $f(\theta_1) = 0$

For θ_2 Using the forward kinematics, we extract the compound angle $\theta_1 + \theta_2$ and get

$$\theta 2 = atan2(y - L1\sin(\theta_1), x - L1\cos(\theta_1)) - \theta_1$$

Bisection Method:

Theory:

Use the standard bisection algorithm to solve $f(\theta_1) = 0$

Execution:

I introduced minimal modifications to the professor's original code to accept the parameters (L1, L2,x,y) and replaced the original test function with the new $f(\theta_1)$. I added a reachability test before running the bisection method to confirm the desired end-effector position within the robot's workspace. After the method converged to a solution for θ1(in radians and degrees), I calculated θ2 (in radians and degrees). I verified the results via forward kinematics, ensuring the correctness of the final solution.

Fixed Point Iteration:

Theory:

Rewrite $f(\theta_1) = 0$ in fixed-point form by defining

$$\theta_1 = \theta_1 - \lambda\ f(\theta_1)$$ (for a small lambda)

From there, via substitution, we get:

$$g(\theta_1) = \theta_1 - \lambda\ ((x - L1\cos(\theta_1))^2 + (y - L1\sin(\theta_1))^2 - L2^2)$$

Execution:

I began by performing the necessary equation transformation, then made minimal adjustments to the professor's original code to allow for additional parameters. This enabled me to apply the method specifically to the function g(θ1). A crucial part of the process was carefully selecting the initial guess, the value of λ, the tolerance (TOL), and the maximum number of iterations (N).

After solving for θ1(in radians and degrees), I computed θ2 (in radians and degrees) to verify the forward kinematics. The model required extensive parameter tuning to reach stability, and the attached results represent a solution as close as possible to a stable outcome.

Newton Method:

Theory:

Using the standard Newton algorithm to solve $f(\theta 1) = 0$,

Where our inverse kinematics problem:

$$f(\theta_1) = (x - L1cos\theta_1)^2 + (y - L1sin\theta_1)^2 - L_2 2.$$

Newton's method for guesses:

$$\theta^{(n+1)} = \theta_1^n - \frac{f(\theta_1^n)}{Df(\theta_1^n)}$$

Derivative given by:

$$Df(\theta_1) = 2L_1 sin\theta_1 (x - L_1 cos\theta_1) - 2L_1 cos\theta_1 (y - L_1 sin\theta_1)$$

Execution:

I made targeted modifications to the professor's original code, adapting it to the inverse kinematics problem by replacing the generic test function and its derivative with problem-specific functions. I then introduced the necessary manipulator parameters, end-effector target, and the convergence criteria (tolerance, maximum iterations, initial guesses) for the Newton method. To confirm feasibility, I added a quick check to ensure the target lies within the robot's workspace. Finally, I ran Newton's method to solve for θ1 (in radians and degrees), computed θ2 (in radians and degrees), and verified the forward kinematics to confirm a valid solution.

Conclusions

This project implemented and analyzed numerical methods for solving the inverse kinematics problem of a two-link robotic manipulator. The three methods—Bisection Method, Fixed-Point Iteration, and Newton's Method—were applied to solve for the joint angles θ1 and θ2 given a target (x,y) end-effector position. The results obtained for each method provide insight into their relative efficiency, accuracy, and reliability.

Key Observations from the Results:

Bisection Method:

- The bisection method successfully computed θ1=1.57082 radians (≈90.001°) after 17 iterations, demonstrating robust convergence.
- The computed θ2=−1.57082 radians (≈-90.001°) were validated using forward kinematics, yielding an end-effector position very close to the desired target (x=1.0,y=1.0).
- This method is slow but highly reliable, especially when a valid interval containing the root is known.

Fixed-Point Iteration:

- The fixed-point iteration method struggled with convergence despite reaching $\theta 1=1.57695$ radians (≈90.35°) after 200 iterations.
- The computed end-effector position ($x=0.9938, y=1.0000$) showed small errors, indicating slow and unstable convergence.
- The method's effectiveness was highly dependent on the choice of the relaxation parameter λ, which affected stability.

Newton's Method:

- Newton's method was the most efficient, computing $\theta 1=1.57079$ radians (≈90.00°) in only 3 iterations.
- The computed $\theta 2=-1.57079$ radians (≈-90.00°) was validated using forward kinematics, yielding an almost perfect match with the target position.
- However, Newton's method is sensitive to the initial guess and may fail if the derivative is close to zero.

Overall, this project demonstrates the importance of choosing an appropriate numerical method based on the problem itself and its own constraints.

References

Blakie, D. (n.d.). Advanced Python coding. NYU, Class Notes.

MAT 5003. (n.d.). Class notes and codes. Course Materials.

Python Software Foundation. (2024). Python tutorial: Control flow tools. Retrieved from https://docs.python.org/3/tutorial/controlflow.html#arbitrary-argument-lists

Sauer, T. (2024). Numerical Analysis (3rd ed.). Pearson.

Wikipedia contributors. (2024). Fixed-point iteration. Wikipedia, The Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Fixed-point_iteration

PythonReferences.com. (n.d.). Python coding resources. Retrieved from https://pythonreferences.com

https://demonstrations.wolfram.com/ForwardAndInverseKinematicsForTwoLinkArm

Appendix – Computational Data and Code.

Bisection method

February 11, 2025

[17]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
```

[18]:
```
def bisection(f, a, b, TOL, *args):
    if np.sign(f(a, *args)) * np.sign(f(b, *args)) > 0:
        print('f(a)*f(b) < 0 not satisfied')
        return None  # Stop execution if the sign condition is not met
    n = 1
    fa = f(a, *args)
    fb = f(b, *args)
    while np.abs(a - b) > TOL:
        c = (a + b) / 2.0
        fc = f(c, *args)
        n += 1
        if np.sign(fc) * np.sign(fa) < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc
    c = (a + b) / 2.0
    print('The final interval [', a, b, '] contains a root')
    print('Approximate root', c, 'has been obtained in', n, 'steps')
    return c
```

[19]:
```
# f_theta1 represents the equation:
# (x - L1*cos(theta1))^2 + (y - L1*sin(theta1))^2 - L2^2 = 0
def f_theta1(theta1, L1, L2, x, y):
    return (x - L1 * math.cos(theta1))**2 + (y - L1 * math.sin(theta1))**2 -␣
 ↪L2**2
```

[20]:
```
# defining the reachability function
def is_reachable(L1, L2, x, y):
    r = math.sqrt(x**2 + y**2)
    return abs(L1 - L2) <= r <= (L1 + L2)
```

[21]:
```
# Define Parameters and Desired End-Effector Position
L1 = 1.0  # Length of link 1
L2 = 1.0  # Length of link 2
x_des = 1.0  # Desired x-coordinate of the hand
y_des = 1.0  # Desired y-coordinate of the hand
TOL = 0.5e-4  # Tolerance for the bisection method
```

[22]:
```
# Test the Reachability and, if reachable, Solve for theta1 using Bisection
if not is_reachable(L1, L2, x_des, y_des):
    print("The desired point (x, y) is outside the reachable workspace.")
else:
    theta1_sol = bisection(f_theta1, 0, math.pi, TOL, L1, L2, x_des, y_des)
    if theta1_sol is not None:
        print("Computed theta1 =", theta1_sol, "radians,", math.
 ↪degrees(theta1_sol), "degrees")
```

```
The final interval [ 1.5707963267948966 1.570844263694518 ] contains a root
Approximate root 1.5708202952447072 has been obtained in 17 steps
Computed theta1 = 1.5708202952447072 radians, 90.00137329101562 degrees
```

[27]:
```
 # Compute theta2 from the computed theta1.
 # We use the rearranged equation:
 # x - L1*cos(theta1) = L2*cos(theta1+theta2)
 # y - L1*sin(theta1) = L2*sin(theta1+theta2)
 # Thus, the compound angle (theta1 + theta2) can be computed with atan2:
theta_sum = math.atan2(y_des - L1 * math.sin(theta1_sol), x_des - L1 * math.
 ↪cos(theta1_sol))
theta2_sol = theta_sum - theta1_sol
print("Computed theta2 =", theta2_sol, "radians,", math.degrees(theta2_sol),␣
 ↪"degrees")
```

```
Computed theta2 = -1.5708202949574708 radians, -90.00137327455819 degrees
```

[31]:
```
 # Verification via Forward Kinematics
def forward_kinematics(L1, L2, theta1, theta2):
        x = L1 * math.cos(theta1) + L2 * math.cos(theta1 + theta2)
        y = L1 * math.sin(theta1) + L2 * math.sin(theta1 + theta2)
        return x, y
x_fk, y_fk = forward_kinematics(L1, L2, theta1_sol, theta2_sol)
    print("Forward kinematics result: x =", x_fk, ", y =", y_fk)
    print("Desired position: x =", x_des, ", y =", y_des)
```

```
Forward kinematics result: x = 0.9999760315501917 , y = 0.9999999999999931
Desired position: x = 1.0 , y = 1.0
```

2_Fixed point iteration

February 11, 2025

0.1 Fixed point iteration

[38]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
```

[39]:
```
def fixedp(g, x0, TOL, N, *args):
    # Fixed point algorithm (modified minimally to allow extra parameters)
    e = 1
    i = 0
    xi = []
    while (e > TOL and i < N):
        x = g(x0, *args)  # Call g with extra arguments
        e = np.abs(x0 - x)  # Compute error at the current step
        x0 = x
        xi.append(x0)  # Save the current iterate
        i = i + 1
    if e < TOL:
        print('Approximate fixed point', x0)
    else:
        print('The fixed point iteration diverges')
    return (xi, i)
```

[40]:
```
def g_fixed(theta1, L1, L2, x, y, lam=0.01):
    # Fixed point function for inverse kinematics:
    # theta1 = theta1 - lam * [ (x - L1*cos(theta1))^2 + (y - L1*sin(theta1))^2
 ↪- L2^2 ]
    return theta1 - lam * ((x - L1 * math.cos(theta1))**2 + (y - L1 * math.
 ↪sin(theta1))**2 - L2**2)
```

[41]:
```
# Manipulator parameters and desired position
L1 = 1.0        # Length of link 1
L2 = 1.0        # Length of link 2
x_des = 1.0     # Desired x-coordinate
y_des = 1.0     # Desired y-coordinate
TOL = 1e-5      # Tolerance for convergence
N = 200          # Maximum iterations
lam = 0.001      # Relaxation parameter
```

[42]:
```
initial_guess = 1.58
```

[43]:
```
xi, iterations = fixedp(g_fixed, initial_guess, TOL, N, L1, L2, x_des, y_des,␣
 ↪lam)
print("Fixed point iterates:")
for idx, val in enumerate(xi):
    print(f"Iteration {idx+1}: {val}")
theta1_sol = xi[-1]
print("Final approximation for theta1:", theta1_sol, "radians,", math.
 ↪degrees(theta1_sol), "degrees")
```

```
The fixed point iteration diverges
Fixed point iterates:
Iteration 1: 1.5799815082064599
Iteration 2: 1.5799630537349818
Iteration 3: 1.5799446365095628
Iteration 4: 1.5799262564543577
Iteration 5: 1.5799079134936782
Iteration 6: 1.5798896075519928
Iteration 7: 1.5798713385539267
Iteration 8: 1.5798531064242611
Iteration 9: 1.579834911087933
Iteration 10: 1.5798167524700348
Iteration 11: 1.5797986304958143
Iteration 12: 1.5797805450906734
Iteration 13: 1.5797624961801693
Iteration 14: 1.579744483690013
Iteration 15: 1.5797265075460691
Iteration 16: 1.579708567674356
Iteration 17: 1.579690664001045
Iteration 18: 1.5796727964524604
Iteration 19: 1.5796549649550788
Iteration 20: 1.5796371694355293
Iteration 21: 1.5796194098205925
Iteration 22: 1.5796016860372006
Iteration 23: 1.5795839980124373
Iteration 24: 1.5795663456735367
Iteration 25: 1.579548728947884
Iteration 26: 1.579531147763014
Iteration 27: 1.5795136020466118
Iteration 28: 1.5794960917265122
Iteration 29: 1.579478616730699
Iteration 30: 1.5794611769873048
Iteration 31: 1.5794437724246113
Iteration 32: 1.5794264029710483
Iteration 33: 1.5794090685551936
Iteration 34: 1.5793917691057726
Iteration 35: 1.5793745045516583
```

```
Iteration 36: 1.5793572748218703
Iteration 37: 1.5793400798455755
Iteration 38: 1.579322919552087
Iteration 39: 1.5793057938708641
Iteration 40: 1.5792887027315115
Iteration 41: 1.57927164606378
Iteration 42: 1.579254623797565
Iteration 43: 1.5792376358629074
Iteration 44: 1.5792206821899921
Iteration 45: 1.5792037627091486
Iteration 46: 1.5791868773508504
Iteration 47: 1.5791700260457142
Iteration 48: 1.5791532087245006
Iteration 49: 1.5791364253181128
Iteration 50: 1.579119675757597
Iteration 51: 1.5791029599741415
Iteration 52: 1.5790862778990773
Iteration 53: 1.5790696294638769
Iteration 54: 1.579053014600154
Iteration 55: 1.579036433239664
Iteration 56: 1.5790198853143032
Iteration 57: 1.579003370756108
Iteration 58: 1.5789868894972559
Iteration 59: 1.5789704414700636
Iteration 60: 1.5789540266069884
Iteration 61: 1.578937644840626
Iteration 62: 1.5789212961037122
Iteration 63: 1.5789049803291213
Iteration 64: 1.5788886974498657
Iteration 65: 1.5788724473990967
Iteration 66: 1.5788562301101032
Iteration 67: 1.5788400455163116
Iteration 68: 1.5788238935512862
Iteration 69: 1.5788077741487279
Iteration 70: 1.5787916872424745
Iteration 71: 1.5787756327665006
Iteration 72: 1.5787596106549164
Iteration 73: 1.5787436208419687
Iteration 74: 1.5787276632620395
Iteration 75: 1.578711737849646
Iteration 76: 1.578695844539441
Iteration 77: 1.5786799832662115
Iteration 78: 1.5786641539648794
Iteration 79: 1.5786483565705005
Iteration 80: 1.5786325910182646
Iteration 81: 1.578616857243495
Iteration 82: 1.578601155181649
Iteration 83: 1.578585484768316
```

```
Iteration 84: 1.5785698459392188
Iteration 85: 1.5785542386302125
Iteration 86: 1.5785386627772844
Iteration 87: 1.5785231183165538
Iteration 88: 1.5785076051842715
Iteration 89: 1.5784921233168199
Iteration 90: 1.5784766726507125
Iteration 91: 1.5784612531225932
Iteration 92: 1.578445864669237
Iteration 93: 1.5784305072275489
Iteration 94: 1.5784151807345639
Iteration 95: 1.5783998851274466
Iteration 96: 1.5783846203434913
Iteration 97: 1.5783693863201211
Iteration 98: 1.5783541829948886
Iteration 99: 1.5783390103054744
Iteration 100: 1.5783238681896876
Iteration 101: 1.5783087565854657
Iteration 102: 1.5782936754308736
Iteration 103: 1.5782786246641038
Iteration 104: 1.5782636042234763
Iteration 105: 1.578248614047438
Iteration 106: 1.5782336540745625
Iteration 107: 1.5782187242435497
Iteration 108: 1.578203824493226
Iteration 109: 1.5781889547625434
Iteration 110: 1.5781741149905797
Iteration 111: 1.5781593051165381
Iteration 112: 1.5781445250797472
Iteration 113: 1.5781297748196599
Iteration 114: 1.578115054275854
Iteration 115: 1.5781003633880317
Iteration 116: 1.5780857020960193
Iteration 117: 1.5780710703397667
Iteration 118: 1.5780564680593474
Iteration 119: 1.5780418951949584
Iteration 120: 1.5780273516869194
Iteration 121: 1.5780128374756732
Iteration 122: 1.5779983525017849
Iteration 123: 1.5779838967059416
Iteration 124: 1.577969470028953
Iteration 125: 1.5779550724117504
Iteration 126: 1.577940703795386
Iteration 127: 1.577926364121034
Iteration 128: 1.5779120533299886
Iteration 129: 1.577897771363666
Iteration 130: 1.5778835181636015
Iteration 131: 1.5778692936714518
```

```
Iteration 132: 1.5778550978289927
Iteration 133: 1.57784093057812
Iteration 134: 1.5778267918608488
Iteration 135: 1.5778126816193139
Iteration 136: 1.5777985997957682
Iteration 137: 1.5777845463325841
Iteration 138: 1.577770521172252
Iteration 139: 1.5777565242573803
Iteration 140: 1.577742555530696
Iteration 141: 1.5777286149350433
Iteration 142: 1.5777147024133837
Iteration 143: 1.5777008179087963
Iteration 144: 1.577686961364477
Iteration 145: 1.5776731327237385
Iteration 146: 1.5776593319300096
Iteration 147: 1.5776455589268357
Iteration 148: 1.5776318136578777
Iteration 149: 1.577618096066913
Iteration 150: 1.5776044060978338
Iteration 151: 1.5775907436946477
Iteration 152: 1.5775771088014772
Iteration 153: 1.57756350136256
Iteration 154: 1.5775499213222477
Iteration 155: 1.5775363686250066
Iteration 156: 1.5775228432154167
Iteration 157: 1.5775093450381723
Iteration 158: 1.5774958740380807
Iteration 159: 1.5774824301600627
Iteration 160: 1.5774690133491522
Iteration 161: 1.5774556235504962
Iteration 162: 1.5774422607093537
Iteration 163: 1.5774289247710964
Iteration 164: 1.5774156156812082
Iteration 165: 1.5774023333852847
Iteration 166: 1.5773890778290331
Iteration 167: 1.5773758489582725
Iteration 168: 1.5773626467189326
Iteration 169: 1.5773494710570541
Iteration 170: 1.577336321918789
Iteration 171: 1.577323199250399
Iteration 172: 1.5773101029982568
Iteration 173: 1.5772970331088445
Iteration 174: 1.5772839895287543
Iteration 175: 1.5772709722046876
Iteration 176: 1.5772579810834557
Iteration 177: 1.5772450161119786
Iteration 178: 1.5772320772372854
Iteration 179: 1.5772191644065134
```

```
Iteration 180: 1.577206277566909
Iteration 181: 1.5771934166658261
Iteration 182: 1.577180581650727
Iteration 183: 1.5771677724691815
Iteration 184: 1.577154989068867
Iteration 185: 1.5771422313975683
Iteration 186: 1.5771294994031768
Iteration 187: 1.5771167930336913
Iteration 188: 1.5771041122372171
Iteration 189: 1.5770914569619654
Iteration 190: 1.577078827156254
Iteration 191: 1.5770662227685064
Iteration 192: 1.5770536437472522
Iteration 193: 1.577041090041126
Iteration 194: 1.577028561598868
Iteration 195: 1.5770160583693231
Iteration 196: 1.5770035803014415
Iteration 197: 1.5769911273442776
Iteration 198: 1.5769786994469905
Iteration 199: 1.5769662965588434
Iteration 200: 1.576953918629203
Final approximation for theta1: 1.576953918629203 radians, 90.35280402406998
degrees
```

[44]:
```python
# Compute theta2 using the rearranged forward kinematics:
# theta1 + theta2 = atan2(y_des - L1*sin(theta1), x_des - L1*cos(theta1))
theta_sum = math.atan2(y_des - L1 * math.sin(theta1_sol), x_des - L1 * math.
 ↪cos(theta1_sol))
theta2_sol = theta_sum - theta1_sol
print("Computed theta2:", theta2_sol, "radians,", math.degrees(theta2_sol),␣
 ↪"degrees")
```

```
Computed theta2: -1.5769350767404342 radians, -90.35172446336547 degrees
```

[45]:
```python
def forward_kinematics(L1, L2, theta1, theta2):
    # Forward kinematics: computes (x, y) from theta1 and theta2
    x = L1 * math.cos(theta1) + L2 * math.cos(theta1 + theta2)
    y = L1 * math.sin(theta1) + L2 * math.sin(theta1 + theta2)
    return x, y
```

[46]:
```python
x_fk, y_fk = forward_kinematics(L1, L2, theta1_sol, theta2_sol)
print("Forward kinematics result: x =", x_fk, ", y =", y_fk)
print("Desired position: x =", x_des, ", y =", y_des)
```

```
Forward kinematics result: x = 0.9938424468999224 , y = 0.9999998839800693
Desired position: x = 1.0 , y = 1.0
```

Newton method

February 11, 2025

0.1 Newton method

[26]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
```

[27]:
```
def newton(f, Df, x0, TOL, N):
    '''Approximate solution of f(x)=0 by Newton's method.
    Input:
      f : function, the function f(x)
      Df: function, the derivative f'(x)
      x0: initial guess
      TOL: stopping criteria (tolerance)
      N: maximum number of iterations
    '''
    xn = x0
    for n in range(0, N):
        fxn = f(xn)
        Dfxn = Df(xn)
        if Dfxn == 0:
            print('Zero derivative. No solution found.')
            return None
        xn = xn - fxn / Dfxn
        if abs(fxn / Dfxn) < TOL:
            print('Approximate solution', xn, 'after steps', n)
            return xn
    print('Exceeded maximum iterations. No solution found.')
    return None
```

[28]:
```
def f_inv(theta1):
    return (x_des - L1 * math.cos(theta1))**2 + (y_des - L1 * math.
 ↪sin(theta1))**2 - L2**2
```

[29]:
```
def Df_inv(theta1):
    return 2 * L1 * math.sin(theta1) * (x_des - L1 * math.cos(theta1)) - 2 * L1␣
 ↪* math.cos(theta1) * (y_des - L1 * math.sin(theta1))
```

[30]:
```
# Manipulator parameters and desired position
L1 = 1.0        # Length of link 1
L2 = 1.0        # Length of link 2
x_des = 1.0     # Desired x-coordinate
y_des = 1.0     # Desired y-coordinate

# Reachability test function
def is_reachable(L1, L2, x, y):
    r = math.sqrt(x**2 + y**2)
    return (abs(L1 - L2) <= r <= (L1 + L2))
# Test
if not is_reachable(L1, L2, x_des, y_des):
    print("The desired point (x, y) is outside the reachable workspace.")
else:
    initial_guess = 1.57
    theta1_newton = newton(f_inv, Df_inv, initial_guess, TOL, N)
print("Newton's method result for theta1:", theta1_newton, "radians,", math.
 ↪degrees(theta1_newton), "degrees")
```

```
Approximate solution 1.5707963267948968 after steps 3
Newton's method result for theta1: 1.5707963267948968 radians, 90.00000000000001
degrees
```

[31]:
```
theta_sum = math.atan2(y_des - L1 * math.sin(theta1_newton), x_des - L1 * math.
 ↪cos(theta1_newton))
theta2_newton = theta_sum - theta1_newton
print("Computed theta2:", theta2_newton, "radians,", math.
 ↪degrees(theta2_newton), "degrees")
```

```
Computed theta2: -1.5707963267948968 radians, -90.00000000000001 degrees
```

[32]:
```
def forward_kinematics(L1, L2, theta1, theta2):
    x = L1 * math.cos(theta1) + L2 * math.cos(theta1 + theta2)
    y = L1 * math.sin(theta1) + L2 * math.sin(theta1 + theta2)
    return x, y

x_fk, y_fk = forward_kinematics(L1, L2, theta1_newton, theta2_newton)
print("Forward kinematics result: x =", x_fk, ", y =", y_fk)
print("Desired position: x =", x_des, ", y =", y_des)
```

```
Forward kinematics result: x = 0.9999999999999999 , y = 1.0
Desired position: x = 1.0 , y = 1.0
```