Jansen Mechanical Linkage

The Jansen Mechanical Linkage of Figure 1 is a planar linkage. This is a system with one degree of freedom (moves in one plane). The input crank (highlighted in green) rotates counter-clockwise and transfers movement to the red joint, resulting in a looping motion traced by the blue and red path at the bottom.

Defining Closure Equations:

Given the length of each bar, the relative locations of the two fixed joints, a system of closure equations can be developed to determine the angle of each link and therefore the location of each joint.

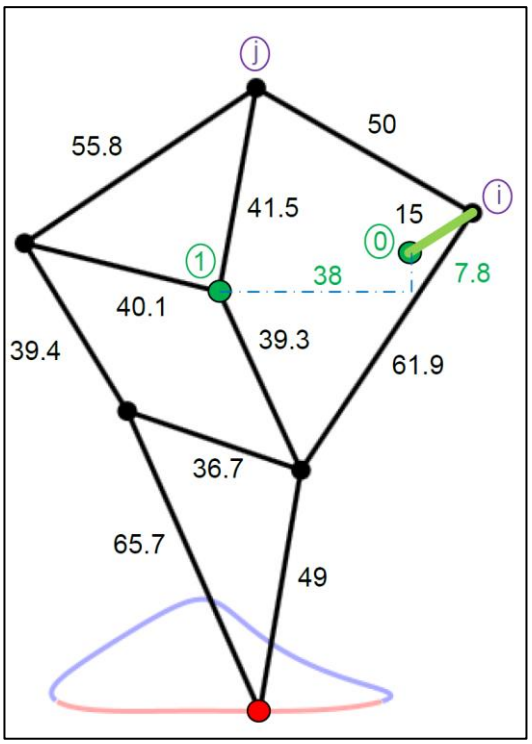

Figure 1: The Jansen Mechanical Linkages

The procedure for developing these closure equations is as such:

Label joints: Self explanatory- joints were labelled alphabetically continuing from i and j to n.

Assign Vectors: Assign each link a vector with direction and length of the link given in Figure 1.

Points	Vector Name	Length
[1, 0]	$\overrightarrow{r_1}$	38.792
[0, i]	$\overrightarrow{r_2}$	15
[i, j]	$\overrightarrow{r_3}$	50
[j, 1]	$\overrightarrow{r_4}$	41.5
[j, k]	$\overrightarrow{r_5}$	55.8
[k, 1]	$\overrightarrow{r_6}$	40.1

Points	Vector Name	Length
[k, l]	$\overrightarrow{r_7}$	39.4
[1, m]	$\overrightarrow{r_8}$	39.3
[l, m]	$\overrightarrow{r_9}$	36.7
[m, i]	$\overrightarrow{r_{10}}$	61.9
[m, n]	$\overrightarrow{r_{11}}$	49
[l, n]	$\overrightarrow{r_{12}}$	65.7

Table 1: Table of pairs of joints and their corresponding vectors

Note: The length of $\overrightarrow{r_1}$ is given by the Pythagorean theorem:

$$\sqrt{38^2 + 7.8^2} = 38.79226727 \quad (1)$$

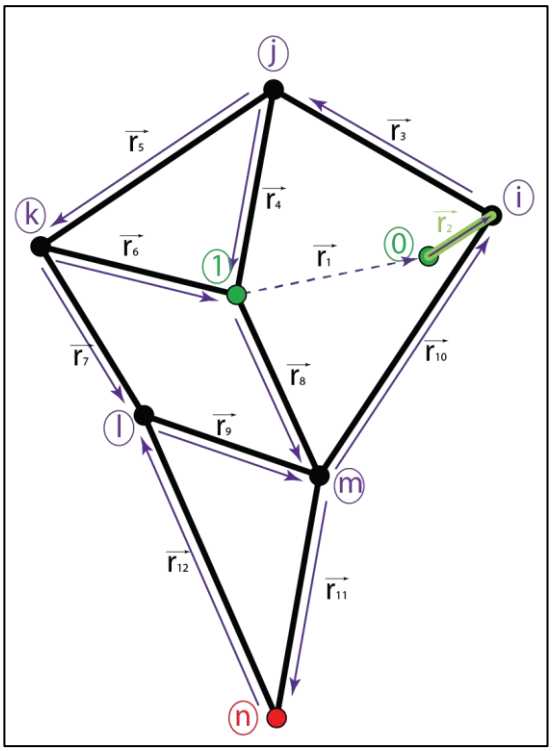

Figure 2: Jansen Linkages with joints and vectors labelled

Identify Loops: Identify closed loops within system. The vectors are followed in a circle until the last vector reaches the start of the first one, adding a positive vector if the arrow is followed, and adding a negative sign if the vector is pointing in the opposite direction of movement. Since the vector loop ends where it begins, the sum of the vectors in the loop adds to zero, yielding 5 equations:

$$1: \overrightarrow{r_1} + \overrightarrow{r_2} + \overrightarrow{r_3} + \overrightarrow{r_4} = 0 \quad (2)$$

$$2: -\overrightarrow{r_4} + \overrightarrow{r_5} + \overrightarrow{r_6} = 0 \quad (3)$$

$$3: \overrightarrow{r_6} + \overrightarrow{r_8} - \overrightarrow{r_9} - \overrightarrow{r_7} = 0 \quad (4)$$

$$4: \overrightarrow{r_1} + \overrightarrow{r_2} - \overrightarrow{r_{10}} - \overrightarrow{r_8} = 0 \quad (5)$$

$$5: \overrightarrow{r_9} + \overrightarrow{r_{11}} + \overrightarrow{r_{12}} = 0 \quad (6)$$

Each vector can be represented in complex form using Euler's formula:

$$\overrightarrow{r_3} = 50e^{i\theta_3} = 50\big(cos\theta_3 + jsin(\theta_3)\big) \quad (7)$$

Hence, each equation has a real and complex version, creating 10 separate equations:

$$f_1: 38 + 15cos(\theta_2) + 50cos(\theta_3) + 41.5cos(\theta_4) \quad (8)$$

$$f_2: 7.8 + 15sin(\theta_2) + 50sin(\theta_3) + 41.5sin(\theta_4) \quad (9)$$

$$f_3: -41.5cos(\theta_4) + 55.8cos(\theta_5) + 40.1cos(\theta_6) \quad (10)$$

$$f_4: -41.5sin(\theta_4) + 55.8sin(\theta_5) + 40.1sin(\theta_6) \quad (11)$$

$$f_5: 40.1cos(\theta_6) - 39.4cos(\theta_7) + 39.3cos(\theta_8) - 36.7cos(\theta_9) \quad (12)$$

$$f_6: 40.1sin(\theta_6) - 39.4sin(\theta_7) + 39.3sin(\theta_8) - 36.7sin(\theta_9) \quad (13)$$

$$f_7: 38 + 15cos(\theta_2) - 39.3cos(\theta_8) - 61.9cos(\theta_{10}) \quad (14)$$

$$f_8: 7.8 + 15sin(\theta_2) - 39.3sin(\theta_8) - 61.9sin(\theta_{10}) \quad (15)$$

$$f_9: 36.7cos(\theta_9) + 49cos(\theta_{11}) + 65.7cos(\theta_{12}) \quad (16)$$

$$f_{10}: 36.7sin(\theta_9) + 49sin(\theta_{11}) + 65.7sin(\theta_{12}) \quad (17)$$

Linearising and Solving Equations:

In this state, the equations aren't suitable for Gaussian elimination due to their nonlinear nature, so they must be estimated. The chosen method of

estimation employed is the multivariate version of the Newton-Raphson method, described as:

$$x_{n+1} = x_n - [\mathbf{J}f(x_n)]^{-1} f(x_n) \tag{18}$$

Where x_n is a vector of the variables present, estimated at the n^{th} iteration, $f(x_n)$ is the matrix of equations with the estimates substituted, and $[\mathbf{J}f(x_n)]^{-1}$ is the inverse of the Jacobian of $f(x_n)$.

To find the Jacobian of the matrix:

$$f_x = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \\ f_{10} \end{bmatrix} \quad x = \begin{bmatrix} \theta_3 \\ \theta_4 \\ \theta_5 \\ \theta_6 \\ \theta_7 \\ \theta_8 \\ \theta_9 \\ \theta_{10} \\ \theta_{11} \\ \theta_{12} \end{bmatrix} \quad \mathbf{J}f_x = \begin{bmatrix} \frac{\partial f_1}{\partial \theta_3} & \frac{\partial f_1}{\partial \theta_4} & \cdots & \frac{\partial f_1}{\partial \theta_{12}} \\ \frac{\partial f_2}{\partial \theta_3} & \frac{\partial f_2}{\partial \theta_4} & \cdots & \frac{\partial f_2}{\partial \theta_{12}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{10}}{\partial \theta_3} & \frac{\partial f_{10}}{\partial \theta_4} & \cdots & \frac{\partial f_{10}}{\partial \theta_{12}} \end{bmatrix} \tag{19}$$

This is:

$$\begin{bmatrix} -50sin(\theta_3) & -41.5sin(\theta_4) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 50cos(\theta_3) & 41.5cos(\theta_4) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 41.5sin(\theta_4) & -55.8sin(\theta_5) & -40.1sin(\theta_6) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -41.5cos(\theta_4) & 55.8cos(\theta_5) & 40.1cos(\theta_6) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -40.1sin(\theta_6) & 39.4sin(\theta_7) & -39.3sin(\theta_8) & 36.7sin(\theta_9) & 0 & 0 & 0 \\ 0 & 0 & 0 & 40.1cos(\theta_6) & -39.4cos(\theta_7) & 39.3cos(\theta_8) & -36.7cos(\theta_9) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39.3sin(\theta_8) & 0 & 61.9sin(\theta_{10}) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -39.3cos(\theta_8) & 0 & -61.9cos(\theta_{10}) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -36.7sin(\theta_9) & 0 & -49sin(\theta_{11}) & -67.5sin(\theta_{12}) \\ 0 & 0 & 0 & 0 & 0 & 0 & 36.7cos(\theta_9) & 0 & 49cos(\theta_{11}) & 67.5cos(\theta_{12}) \end{bmatrix} \tag{20}$$

To implement Newton Raphson effectively, four helper functions were written in Python The first set of helper functions, written in a file called linEqSolvers.py are implementations of Gaussian Elimination and Gauss Seidel iteration:

```
import numpy as np
```

```
# Two functions for solving systems of linear equations: Gaussian Elimination and
Gauss-Seidel Iteration.

# Written by Chukwuebuka Amadi-Obi for EEEN30150 Modelling and Simulation: Major
Project 1

# Created 09/04/2024


# Define function
def gauss_elim(matrix_start, cons, verbose = False):
    """Performs Gaussian elimination (patrial pivoting) on a given system of linear
equations in matrix form.

    Args:
        matrix: A numpy array representing the n*n matrix to be solved. (floats or
complex numbers)
        cons: vector of constants
        verbose: A booelan, set to true if a detailed output of operations is
desired. (defaults to false)

    Returns:
        - A numpy array containing the values for each variable in their respective
index
    """

    #get number of variables
    N = len(matrix_start[0])

    #get number of equations
    M = len(matrix_start)

    matrix = np.zeros((M,N+1), dtype = cons.dtype)

    # add constant vector to end of matrix
    for i in range(M):
        matrix[i] = np.append(matrix_start[i],cons[i])

    if verbose:
        print(f"**************************************************\n\n\
                        Starting matrix:\n{matrix}\n\n\
              **************************************************\n")

    # ------------------------------- Pivoting --------------------------------
#
    #for all variables
```

```
    for i in range(N):
        max_row=i
        #find maximum coefficient of current variable
        for j in range(i,M):
            if abs(matrix[j][i])>abs(matrix[max_row][i]):
                max_row=j

        #pivot by swapping rows
        matrix[[i,max_row]] = matrix[[max_row,i]]

        if verbose:
            print(f"swapped rows {i} and {max_row}: \n{matrix}\n")

        #subtract multiples of selected row from all others
        divisor = matrix[i][i]
        for k in range(i+1,M):
            fact = matrix[k][i]/divisor
            matrix[k]=matrix[k]-(matrix[i]*fact)

        if verbose:
            print(f"subtracted multiples of row {i} from all others:
\n{matrix}\n\n\
                  **************************************************\n")

    # ---------------------------- Back Substitution ----------------------------
#
    #for all variables (working backwards from the bottom)
    for i in range(N-1,-1,-1):
        if matrix[i][i]!=1:     #if not already, divide row by value to get
variable value
            matrix[i]/=matrix[i][i]

        if verbose:
            print(f"Variable {i} is equal to {round(float(matrix[i][-1]),3)}")

        # From all rows above it, subtract multiple of current row to remove
variable
        for j in range(i-1,-1,-1):

            matrix[j]-=matrix[i]*matrix[j][i]

        if verbose and i!=0:
            print(f"Back substituted variable {i} \n{matrix}\n")

    # Select value column and load into result value
```

```
    result = matrix[:,[-1]][:N].reshape((1,N))[:N].reshape((1,N))[0]
    if verbose:
        print(f"\n**************************************************\n\n\
                    Identity matrix found: \n{matrix}\n\n Variable values are:
\n{result}\n")
    return result   #return result list
```

The first function, gaussian_elimination() as described in the docstring, takes the matrix of coefficients and the column matrix of consatnts as input, as well as a Boolean to turn on debug text. It performs Gaussian elimination with partial pivoting, and then returns the solution vector.

It starts by concatenating the column vector on to the coefficient matrix for easier manipulation, and then loops through each of the variable columns. For each one it finds the coefficient of maximum magnitude, pivots by moving it to the top, and then subtracts mutliples of the selected row from the rows below it. The resulting matrix is now in row echelon form.

This makes the next step, substitution easier. The program then starts from the last variable, dividing the whole row by the remaining coefficient, and then subtracting multiples of the selected rows from the upper rows. This continues for all the variables until the matrix achieves reduced row echelon form. In this form, the column vector can be extracted again and then returned as the solution vector.

```
# Define function
def gauss_seidel(A, b, initial_guess, tolerance=1e-9, max_iterations=100,
verbose=False):
    """Performs Gauss-Seidel Iteration on a given system of linear equations in
matrix form.

    Args:
        A : Coefficient matrix.
        b : Constant vector.
        initial_guess : Initial guess for the solution vector.
        tolerance : Tolerance level for convergence. Default is 1e-6.
        max_iterations : Maximum number of iterations allowed. Default is 1000.
```

```
    Returns:
        - A numpy array containing the values for each variable in their respective
index
    """

    if verbose:
        np.set_printoptions(precision=3)

    A_transpose = np.transpose(A)  # Transpose of A for preconditioning
    A = np.dot(A_transpose, A)
    b = np.dot(A_transpose, b)  # Preconditioned constant vector

    x = initial_guess.copy()   #make a copy of the initial guess to avoid modifying
it
    x_new = np.zeros_like(x)
    n = len(x)

    LU = np.tril(A,-1)+np.triu(A,1)

    for i in range(max_iterations):  #iterate until convergence or maximum
iterations reached

        if verbose:
            print(f"\n**********ITERATION: {i+1}**********")

        for j in range(n):

            if verbose:
                print(f"x[{j}] = ({b[j]} + {-1*LU[j]}*{x_new}) divided by
{A[j][j]}",end="")

            x_new[j] = (b[j]+np.sum(np.dot(x_new,-1*LU[j])))/A[j][j]

            if verbose:
                print(f" (Equals {x_new[j]})")

        if (all((x_new - x) < tolerance)):
            if verbose:
                print(f"Tolerance reached, estimated: \n {x_new}")
            return x_new

        x = x_new.copy()

    if (all((x_new - x) < tolerance)):
```

```
        if verbose:
            print(f"Iteration limit reached, estimated: \n {x_new}")
        return x_new
```

The Python implementation of Gauss Seidel as above takes matrix A (coefficient matrix), b (column matrix of constants) and initial_guess (another column matrix of initial guesses) as inputs. The return value is another column matrix of the estimated values within the tolerance.

At the very start of the function, the input matrix A may not be symmetric or positive definite. In this case the function may take very long to converge, if at all. Because of this, at the start of the function it is multiplied by it's transpose to make it both symmetric and positive definite. To avoid skewing the output, the constant column matrix will be multiplied by the transpose too.

$$Ax = b \Rightarrow A^T Ax = A^T b \tag{21}$$

These matrices undergo standard Gauss-Seidel iteration; within each iteration (up to max iterations specified by the user) the function loops through each variable, manipulating the i^{th} equation to isolate the variable, where "i" is the index of the variable selected. Once isolated, the current guesses for the other variables are input into the equation, and a new guess for the variable is reached. This continues until all variables have a new guess and the iteration is complete.

This happens until the change of each variable value between iterations is less than the threshold, at which point the estimates are returned.

The second set of helper functions, implemented in jansen_estimate.py are as follows:

```
import numpy as np
from numpy.linalg import inv, norm
from numpy import cos, sin

# ---------------------------- HELPER FUNCTIONS ---------------------------- #
```

```
def f(theta2, x):
    """
    Calculate value of f(x) for give values of theta2-theta12
    Args:
        theta2: Input angle of link 2 (in radians)
        x: length 11 array of theta variables
    Returns:
        y: value for all equations given variables
    """
    y = np.array([38+15*cos(theta2)+50*cos(x[0])+41.5*cos(x[1]),
           7.8+15*sin(theta2)+50*sin(x[0])+41.5*sin(x[1]),
           -41.5*cos(x[1])+55.8*cos(x[2])+40.1*cos(x[3]),
           -41.5*sin(x[1])+55.8*sin(x[2])+40.1*sin(x[3]),
           40.1*cos(x[3])-39.4*cos(x[4])+39.3*cos(x[5])-36.7*cos(x[6]),
           40.1*sin(x[3])-39.4*sin(x[4])+39.3*sin(x[5])-36.7*sin(x[6]),
           38+15*cos(theta2)-39.3*cos(x[5])-61.9*cos(x[7]),
           7.8+15*sin(theta2)-39.3*sin(x[5])-61.9*sin(x[7]),
           36.7*cos(x[6])+49*cos(x[8])+65.7*cos(x[9]),
           36.7*sin(x[6])+49*sin(x[8])+65.7*sin(x[9])])
    return y

def jf(x):
    """
    Calculate value of the jacobian of f(x) for given values of theta3-theta12
    Args:
        x: length 10 array of theta variables

    Returns:
        y: value for all equations given variables
    """
    y = np.array([[-50*sin(x[0]),-41.5*sin(x[1]),0,0,0,0,0,0,0,0],
            [50*cos(x[0]),41.5*cos(x[1]),0,0,0,0,0,0,0,0],
            [0,41.5*sin(x[1]),-55.8*sin(x[2]),-40.1*sin(x[3]),0,0,0,0,0,0],
            [0,-41.5*cos(x[1]),55.8*cos(x[2]),40.1*cos(x[3]),0,0,0,0,0,0],
            [0,0,0,-40.1*sin(x[3]),39.4*sin(x[4]),-
39.3*sin(x[5]),36.7*sin(x[6]),0,0,0],
            [0,0,0,40.1*cos(x[3]),-39.4*cos(x[4]),39.3*cos(x[5]),-
36.7*cos(x[6]),0,0,0],
            [0,0,0,0,0,39.3*sin(x[5]),0,61.9*sin(x[7]),0,0],
            [0,0,0,0,0,-39.3*cos(x[5]),0,-61.9*cos(x[7]),0,0],
            [0,0,0,0,0,0,-36.7*sin(x[6]),0,-49*sin(x[8]),-67.5*sin(x[9])],
            [0,0,0,0,0,0,36.7*cos(x[6]),0,49*cos(x[8]),67.5*cos(x[9])]])
    return y
```

The first function f() takes the input crank angle as well as the guesses for the other angles and returns the values for each of the 10 equations after substituting them in.

Function jf() is the Jacobian of f, taking in the 10 theta variables and returning the Jacobian after substituting them in. This function doesn't require the crank angle as input because it doesn't appear in the Jacobian matrix.

These helper functions serve the purpose of streamlining the implementation of the ensuing Newton Raphson function. The first function calculates value of the vector of equations for a given set of variables.

The Newton Raphson function is implemented in Python file jansen_estimate.py as below:

```
def jansen_estimate(theta2, xn, solver=0, tolerance = 1e-9, max_iterations = 20,
verbose = False):
    """Estimate roots of multivariate functions f(x)

    Args:
        theta2: Input angle of link 2 (in radians)
        xn: Initial guesses for angles of linkages 3-12 (in radians)
        solver: solve linear equations using Gaussian Elimination or Gauss-Siedel
methods [int 0 or 1 respectively]
        tolerance: Max acceptable value of any f(x) values before iteration ends
        verbose: If true, print debug text to terminal
        max_iterations: Maximum number of iterations before force ending loop

    Returns:
        - A vector of variables at which f(x) has roots
    """

    if verbose: np.set_printoptions(precision=3); print(f"**********INITIAL
GUESS**********\n{xn}\n")
    iteration_count = 0 # int for counting iterations
    while iteration_count < max_iterations: # loop until all iterations are
finished
        if verbose: print(f"**********ITERATION {iteration_count+1}**********")
```

```
        fxn = f(theta2, xn) #calculate residual for guess
        jfxn = jf(xn)   #calculate jacobian for guess

        if np.linalg.norm(fxn) < tolerance:
            if verbose: print(f"Approximate solution is: {xn}\n")
            return xn

        if verbose: print(f"Residual is: {fxn}\n")

        if solver == 0:   #if using Gaussian Elimination
            deltax = gauss_elim(jfxn, -1*fxn)

        if solver == 1:
            deltax = gauss_seidel(jfxn, -1*fxn, xn, tolerance)

        xn+=deltax

        iteration_count+=1  #incease iteration count
```

With the function designed, the user inputs a crank angle initial guess for the other angles of the system (both in radians) as well as an integer (0 or 1) to select to solve the linear equations with Gaussian elimination or Gauss-Seidel iteration respectively. The function will perform Newton Raphson iteration to compute an estimate until the default accuracy is achieved, or the iteration limit is reached. This implementation uses a somewhat arbitrary default accuracy of 10^{-9} and the recommended iteration limit of 20. If desired, a different accuracy and iteration limit can be set, as well as debug output toggled.

This function simply uses Newton Raphson iteration to linearise the closure equations, and then Gaussian elimination or Gauss-Seidel iteration to solve the linearised equations. This repeats until the norm of the error matrix is below the threshold, or the iteration limit is reached.

Now that these functions have been implemented, they can be used to estimate the joint locations at each input angle.

Determining Joint Locations and Animation:

Given that no initial guesses are provided in the source material, measuring the angles digitally from the image of the linkages provided will be sufficient, given that it is drawn roughly to scale:

Vector Name	Angle (rad)
$\overrightarrow{r_1}$	0.202
$\overrightarrow{r_2}$	0.576
$\overrightarrow{r_3}$	2.618
$\overrightarrow{r_4}$	-1.745
$\overrightarrow{r_5}$	-2.548
$\overrightarrow{r_6}$	-0.244

Vector Name	Angle (rad)
$\overrightarrow{r_7}$	-1.030
$\overrightarrow{r_8}$	-1.134
$\overrightarrow{r_9}$	-0.332
$\overrightarrow{r_{10}}$	0.977
$\overrightarrow{r_{11}}$	-1.745
$\overrightarrow{r_{12}}$	1.99

Table 2: Names of each vector and it's angle relative to positive x

Note that the angle for $\overrightarrow{r_2}$ is independent and will vary. The angle present in the table is estimated from the image.

With the angles of each linkage estimated with sufficient accuracy, and the length of each linkage, the system can be defined for any crank angle desired.

By assessing Figure 3, equations can be derived for the location of each of the points, given we now know the length and orientation of each of the vectors.

The x and y components are extracted using trigonometric ratios, then these components are added together to reach the point depending on which vectors the point is at the end of.

Points	X co-ordinate	Y co-ordinate
0	0	0
1	-38	-7.8
i	$15\cos(\theta_2)$	$15\sin(\theta_2)$
j	$-38-41.5\cos(\theta_4)$	$-7.8-41.5sin(\theta_4)$
k	$-38-40.1\cos(\theta_6)$	$-7.8-40.1sin(\theta_6)$
l	$-38-40.1\cos(\theta_6)$ $+39.4cos(\theta_7)$	$-7.8-40.1sin(\theta_6)$ $+39.4sin(\theta_7)$
m	$-38+39.3\cos(\theta_8)$	$-7.8+39.3\sin(\theta_8)$
n	$-38+39.3\cos(\theta_8)$ $+49cos(\theta_{11})$	$-7.8+39.3\sin(\theta_8)$ $+49cos(\theta_{11})$

Table 3: Each point in system relative to point 0

Implementing this in Python and then completing these calculations for a range of angles between 0 and 2π radians allows us to estimate the location of each point within the system for the range of crank angles in a rotation.

The Python implementation in jansen_estimate.py is as follows:

```
# ---------------------- JOINT ANGLES AND ANIMATION ---------------------- #
inital_guesses = np.array([2.618,  #initial estimates for angles of all vectors
                    -1.745,
                    -2.548,
                    -0.244,
                    -1.030,
                    -1.134,
                    -0.332,
                    0.977,
                    -1.745,
                    1.99])

crank_angles = np.linspace(0,2*np.pi,100)   #range of 100 crank angles between 0
and 2pi rads
link2 ,link3 ,link4 ,link5 ,link6 ,link7 ,link8 ,link9 ,link10 ,link11 ,link12 =
[],[],[],[],[],[],[],[],[],[],[]   #lists to hold representations of each linkage

trace = ([],[]) #list to hold x and y co-ordinates of each point on trace

for iter in range(len(crank_angles)):     #for all crank angles
    theta2 = crank_angles[iter]
```

```
    thetas= jansen_estimate(theta2,inital_guesses,0)     #estimate other linkage
angles using Newton Raphson and Gaussian elimination

    zero=[0,0]             # create two lists for x and y co-ords of each point
using angle estimates
    one=[-38,-7.8]
    i=[15*cos(theta2),15*sin(theta2)]
    j=[-38-41.5*cos(thetas[1]),-7.8-41.5*sin(thetas[1])]
    k=[-38-40.1*cos(thetas[3]),-7.8-40.1*sin(thetas[3])]
    l=[-38-40.1*cos(thetas[3])+39.4*cos(thetas[4]),-7.8-
40.1*sin(thetas[3])+39.4*sin(thetas[4])]
    m=[-38+39.3*cos(thetas[5]),-7.8+39.3*sin(thetas[5])]
    n=[-38+39.3*cos(thetas[5])+49*cos(thetas[8]),-
7.8+39.3*sin(thetas[5])+49*sin(thetas[8])]

    link2.append([[zero[0],i[0]],[0,0],[zero[1],i[1]]])     # for each linkage add
proper co-ords to list
    link3.append([[i[0],j[0]],[0,0],[i[1],j[1]]])
    link4.append([[j[0],one[0]],[0,0],[j[1],one[1]]])
    link5.append([[j[0],k[0]],[0,0],[j[1],k[1]]])
    link6.append([[k[0],one[0]],[0,0],[k[1],one[1]]])
    link7.append([[k[0],l[0]],[0,0],[k[1],l[1]]])
    link8.append([[one[0],m[0]],[0,0],[one[1],m[1]]])
    link9.append([[l[0],m[0]],[0,0],[l[1],m[1]]])
    link10.append([[i[0],m[0]],[0,0],[i[1],m[1]]])
    link11.append([[m[0],n[0]],[0,0],[m[1],n[1]]])
    link12.append([[n[0],l[0]],[0,0],[n[1],l[1]]])

    trace[0].append(n[0])       # append x and y co-ords of pointon trace for
current crank angle
    trace[1].append(n[1])

fig = plt.figure()  #set up figure
ax = fig.add_subplot(projection='3d')  #Set up 3d projecction plot


ax.set_xlim(-140,50)        #set x and z limits
ax.set_zlim(-100,50)
lw = 4      #set line width
lc = "grey" #set line colour

def update(frame):      # update function to be called repeatedly
    for art in list(ax.lines):      #remove all art
        art.remove()
    ax.plot3D(*link2[frame],color = lc,linewidth=lw)    #draw all linkages
    ax.plot3D(*link3[frame],color = lc,linewidth=lw)
```

```
    ax.plot3D(*link4[frame],color = lc,linewidth=lw)
    ax.plot3D(*link5[frame],color = lc,linewidth=lw)
    ax.plot3D(*link6[frame],color = lc,linewidth=lw)
    ax.plot3D(*link7[frame],color = lc,linewidth=lw)
    ax.plot3D(*link8[frame],color = lc,linewidth=lw)
    ax.plot3D(*link9[frame],color = lc,linewidth=lw)
    ax.plot3D(*link10[frame],color = lc,linewidth=lw)
    ax.plot3D(*link11[frame],color = lc,linewidth=lw)
    ax.plot3D(*link12[frame],color = lc,linewidth=lw)
    ax.plot3D(trace[0],np.zeros(len(trace[0])),trace[1], color =
"#b5b5b5",zorder=1)   #draw trace of bottom joint

ani = anim.FuncAnimation(fig, update, frames = len(link2), interval = 30)   #
animation function, input figure to draw on, funciton to call, length of animaiton
and time interval between frames
plt.show()  #show animation
```

The initial guesses measured from the image are initialised as a list, and a range of crank angles from 0 to 2π radians are used to determine the angle of all the other linkages. The list is iterated over and for each crank angle, the points of each joint are recorded and used to calculate both ends of each joint.

At the end of this loop, each linkage has a list of start and end points for each angle of the crank. By iterating over all of these lists at the same time and displaying them in 3D, the full image of the system can be seen, with the trace of the bottom joint drawn from the recorded joint locations. Since the system is flat, the third co-ordinate of all the points is set to 0.

Curve Fitting of Upper and Lower Traces

By separating the trace of the lower points into two separate curves, we can use curve fitting and Fermat's theorem to fit a polynomial curve to these traces. The Python implementation of this curve fitting algorithm in jansen_estimate.py is as follows:

```
import numpy as np
from linEqSolvers import gauss_elim
```

```
def curve_fit(x, y):
    """
    Curve fitting function that minimizes mean square error using Fermat's Theorem.

    Args:
        x (numpy array): Independent variable values.
        y (numpy array): Dependent variable values.

    Returns:
        - A list of 5th order polynomial coefficients.

          In the form a + bx +cx^2 + dx^3, ex^4, fx^5, function
          returns [a, b, c, d, e, f]
    """

    x=x.astype("float64")   # Convert data to large floats for easier manipulation
    y=y.astype("float64")

    lin_eq_mat = np.zeros([6,6])    #array for linear equation coefficient
    cons_mat = np.zeros([6,1])      #array for constants of linear equations

    for i in range(6):      # create 6 linear equations using Fermat's theorem
        k = sum(y*(x**i))
        k0 = sum(x**i)
        k1 = sum(x**(1+i))
        k2 = sum(x**(2+i))
        k3 = sum(x**(3+i))
        k4 = sum(x**(4+i))
        k5 = sum(x**(5+i))

        lin_eq_mat[i] = [k5,k4,k3,k2,k1,k0]     # Update values in coefficient and
constant arrays
        cons_mat[i] = k

    return gauss_elim(lin_eq_mat,cons_mat)  # return answers in polynomial form
```

This function takes two arrays, one of the x and one of the y co-ordinates of each point on a given plane. The function then uses curve fitting with Fermat's theorem to fit a 5^{th} order polynomial to the points given, minimizing the mean square error.

Note that a 5^{th} order polynomial was chosen to reach an even lower mean square error of the final curve, lower order polynomials are limited in terms

of their accuracy due to their lower amount of inflection points, resulting in an inability to follow complex curves properly.

This function was implemented in curve_fit.py to follow the curve of the traces found previously:

```
# ------------------ CURVE FITTING OF UPPER AND LOWER TRACE ------------------ #

u_trace_x = np.array(trace[0][32:72])   # Split trace into upper and lower
components
u_trace_y = np.array(trace[1][32:72])

l_trace_x = np.append(trace[0][71:100],trace[0][0:33])
l_trace_y = np.append(trace[1][71:100],trace[1][0:33])

x_range = np.linspace(l_trace_x[0],l_trace_x[-1],200)   # get range of x values to
plot curves over
u_fit_curve = np.polyval(curve_fit(u_trace_x,u_trace_y),x_range)    # fit curves to
upper and lower trace
l_fit_curve = np.polyval(curve_fit(l_trace_x,l_trace_y),x_range)

plt.plot(x_range,u_fit_curve)       #plot curves and original trace
plt.plot(x_range,l_fit_curve)
plt.scatter(u_trace_x,u_trace_y, color = "blue",s=4, alpha=0.1875)
plt.scatter(l_trace_x,l_trace_y, color = "red",s=4, alpha=0.1875)

plt.show()  #display fit curves with traces
```

The first step is to separate the 100 points on the trace into two different curves, the upper and lower. This was done manually by finding the index of the maximum and minimum x values on the trace, and splitting it there. The result is four arrays of co-ordinates, an x and y array for each line.

After this, the range of 200 x values is selected (also from the maximum and minimum points) and used as the input values for the polyval() NumPy function. This function computes the corresponding y value of a polynomial. The polynomial used in both curves is the result of the curve fitting function implemented previously.

The result of this operation is 200 corresponding y values for the upper and lower trace each, with the y values lying on the points of the fit line. Using matplotlib, the polynomials can be plotted alongside the points:

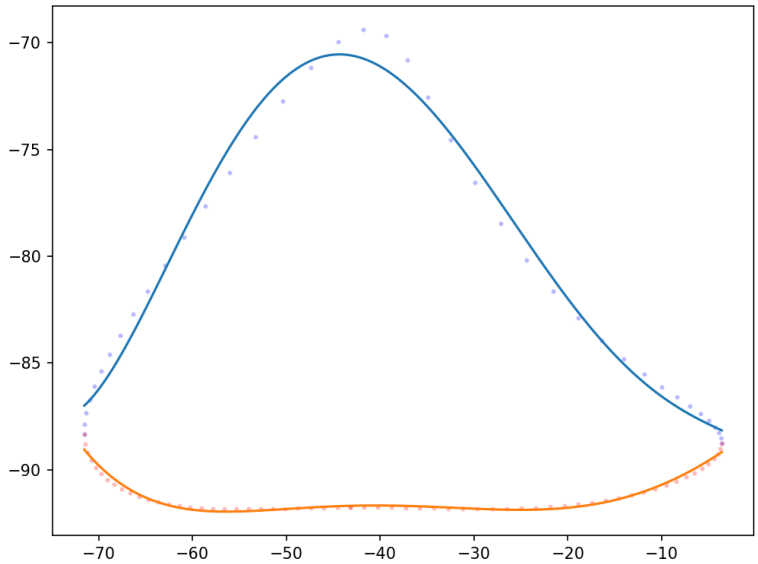

Figure 3: Lines of best fit (blue and orange) with their corresponding points (lighter markers)

By printing out the result of the curve fitting function, the equation of the lines can be found:

$$upper\ curve = -2.211 \times 10^{-7}x^5 - 3.366 \times 10^{-5}x^4 - 1.373 \times 10^{-3}x^3 - 9.691 \times 10^{-3}x^2 - 2.235 \times 10^{-1}x - 88.89 \quad (22)$$

$$lower\ curve = -5.063 \times 10^{-8}x^5 - 6.181 \times 10^{-6}x^4 - 1.482 \times 10^{-4}x^3 + 7.187 \times 10^{-3}x^2 + 3.503 \times 10^{-1}x - 88.0 \quad (23)$$

Reflections:

Overdetermined System: After determining the equations for each point's co-ordinates as a function of the angles in the structure, not all of the angles were used; θ_3 and θ_5.

This means the system can be fully defined with just 8 equations. This is because one angle can be left unknown within a closure equation, and it will remain with a single answer.

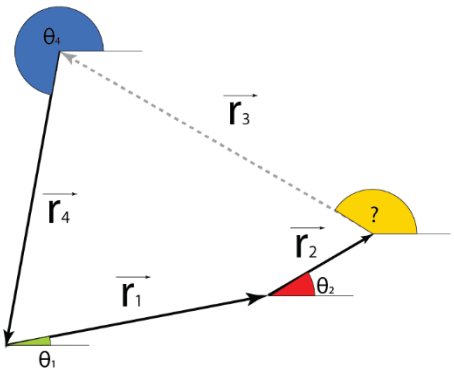

Figure 3: θ_3 is unknown in this loop but vector $\overrightarrow{r_3}$ can still be found

This means one of the 5 loop equations could have been left out without effecting the operations that followed.

A system with too many equations (like the 10 equations in this report) is called "overdetermined" meaning it has more equations than unknowns. An overdetermined system can still have a solution if all equations are consistent, so it isn't too much of an issue. A foreseeable downside to having an overdetermined system is the extra computation resources required to estimate solutions to the extra equations.

Animation Rendering: The efficiency with which the system was animated could be improved. The matplotlib animation module is used to streamline the rendering and displaying of animations, but due to the time limitations this wasn't implemented entirely properly.

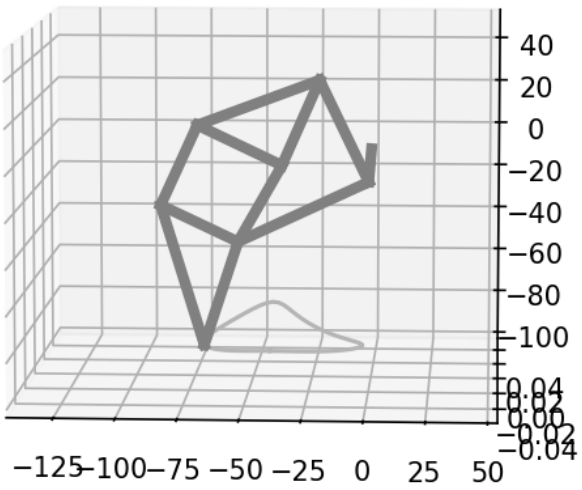

Figure 3: a frame from the 3D animation of the linkage system with the trace of point n (light grey)

As a result the "animation" happens by, clearing the plot and re-drawing it with the next frame at a high frequency. The downside to this is that items that don't need to be animated, such as the trace at the bottom must be redrawn as well, spending more resources and slowing the frame rate.

Initial Angle Estimation: Estimating the initial angles from the photo given in the assignment is the best available method to provide an initial guess.

The photo is of the system with an input crank angle of 0.576 radians, so the initial guess will be the most accurate for that angle, but the same initial guess is sufficient accuracy for all input crank angles, as the Newton Raphson function has no problem converging for all angles.