Franklin W. Olin College of Engineering

MTH3199 Applied Math for Engineers – Fall 2025
Assignment 2: Strandbeest Simulation

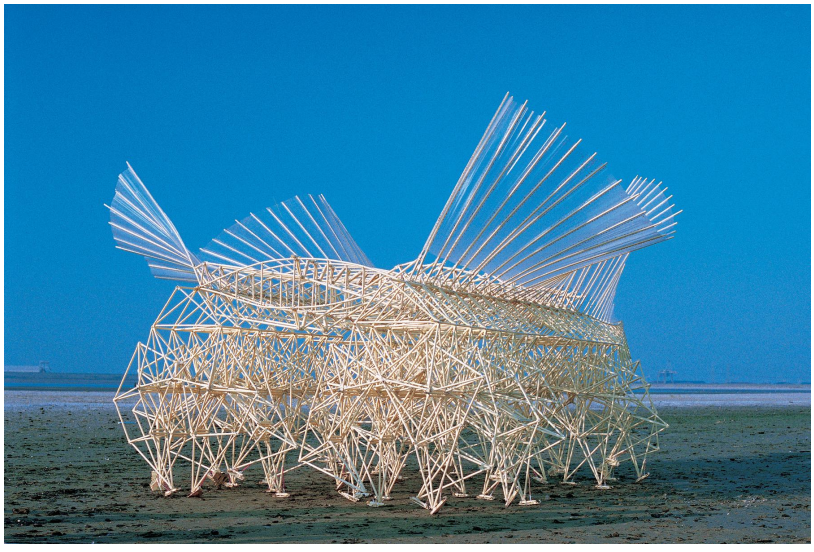
Assigned: Tuesday, September 23, 2025.
Lab Report Due: Thursday, October 2nd, 2025 (11:59 PM EST).

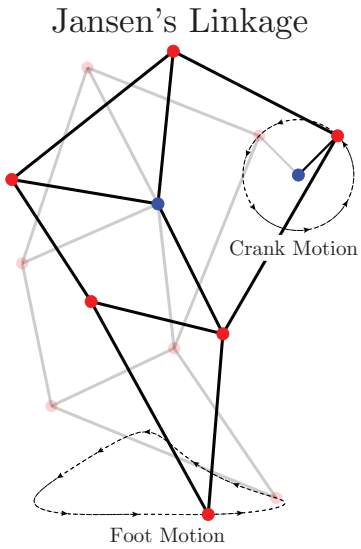
Online Resources

- Description of the Jacobian: Wikipedia, Khan Academy, Wolfram MathWorld.
- Newton's method in multiple dimensions: Glyn A. Holton,
- Jansen's linkage: Wikipedia.

Overview

We have spent a lot of time over the past few weeks developing some powerful numerical tools and practicing how to apply them to some toy problems. Now that we have a multidimensional root-finding algorithm at our disposal, let's have some fun with it. I have chosen linkage mechanisms to be our system of focus for the remainder of this module. Linkages have interesting math, they look cool (which means your video deliverables will look awesome), and root-finding is the perfect tool for simulating their behavior. Specifically, we will be simulating Jansen's linkage (shown below). Theo Jansen is an artist who has designed many neat wind-powered walking sculptures called Strandbeests.





Each leg of a Strandbeest consists of a Jansen linkage that is driven by a rotating crank. This linkage is specifically designed to generate a natural walking motion. Over the course of this assignment, we will walk through the process of simulating this linkage, step by step.

Oftentimes, one of the most difficult challenges in modelling and simulation is figuring out how to ask the right question. We currently have a tool that is very effective at giving us a certain type of answer (i.e. finding the value of X for which $f(X) = 0$). In order to simulate the linkage, we will need to figure out how to represent our task using a format that our root-finding algorithm can process. If we can convert our problem into a question of the form "what is the root of function $f(X)$?", then we can hand this problem over to our root-finder, and it will (hopefully) give us an answer that we can then decode to learn more about the system.

As such, you will be spending much of your time in this assignment learning how to express the task of finding the linkage state as a constraint satisfaction problem. We will represent the linkage configuration using a set of

variables and then identify the mathematical constraints that these variables must satisfy. It is then straightforward to manipulate the constraint satisfaction problem into a root-finding problem, which we can then hand to our solver.

Just as a side-note, I would like to add a few words of caution: "when all you have is a hammer, everything starts to look like a nail". You should not approach a modelling and simulation problem with the assumption that you will be using a particular numerical/analytical/modelling tool. Each tool has its strengths and limitations, and it's important to learn how to choose the right tool for the job. In this case, since I'm giving you a "canned problem", we are skipping that part, but you should still keep it in mind. What I'm trying to say is: don't assume every problem can be solved using root-finding (though it is still extremely useful!).

Instructions

Day 7 (Tuesday, September 23rd) activity. Please complete before class on Friday, September 26th.

Our goal is to simulate the motion of the linkage. As the crank-shaft rotates, how does the linkage move? How does its shape change? How do we figure out the path that the leg-tip travels or the velocity of the leg-tip? These are the kinds of questions we'd like to answer.

In order to frame this as a root-finding problem, we will begin by figuring out how to mathematically describe the configuration of the linkage with a set of continuous variables. We will then identify the kinematic constraints that govern the linkage, which map directly to a set of equations that our variables must satisfy. Once we have a set of variables and constraint equations, we can then generate a function, $f(X)$, whose root corresponds to a "legal" configuration of the system, given the current angle of the crank. We will then find the root, X_{root}, using multidimensional Newton's method. The value of X_{root} can then be used to describe the motion of the linkage. Once this has been accomplished, we will write a program that generates a visualization of this motion. Finally, we will learn how to use the Jacobian of $f(X)$ to make predictions about the velocity of the leg-tip.

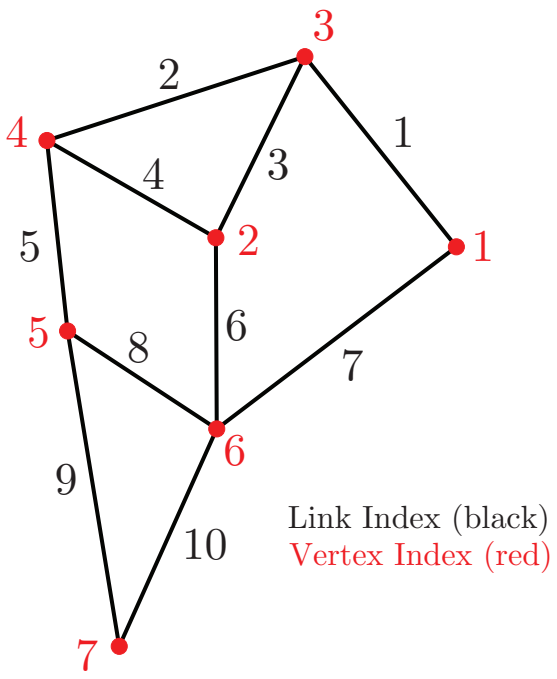
Jansen's linkage primarily consists of two types of objects: **links** and **vertices**. **Links** are essentially just rigid rods that are attached to one another via pin-joints (which we'll refer to as **vertices**). There are other types of joints that linkages can use (ex. sliding joints), though this is not the case for Jansen's linkage. There are many equally valid ways to represent the configuration of the linkage (ex. you could keep track of the position and orientation of each link). For this particular linkage, it is convenient to represent the state using the position of each pin-joint (vertex). Since this is a planar linkage, each position can be described as an x-y coordinate pair, (x_i, y_i), where i is the label of the current pin-joint we are looking at. The combination of these position coordinates will form the set of variables in our constraint satisfaction problem:

$$((x_1, y_1), (x_2, y_2), ..., (x_n, y_n)) \tag{1}$$

Each link will then correspond to a distance constraint on a pair of pin-joints. In addition, we will have a few extra constraints which describe certain vertices being fixed in place. That is the entirety of our constraint satisfaction problem, which we can then solve numerically.

Adjacency Description

In order to simulate the leg linkage mechanism, we need to be consistent with how we keep track of every moving part. This is easily accomplished by assigning labels to each individual vertex and link. An example labelling of the seven vertices and the ten links is provided (feel free to use it!). On closer inspection, we see that the links and vertices form an undirected graph. This is a particularly useful abstraction for representing the relationships between the various objects in the linkage. Each individual link describes an adjacency relationship between a pair of vertices. There are several ways to encode which vertices are connected, including adjacency matrices and adjacency lists. For our purposes, it is most convenient to use an edge list. An edge list E is a two column matrix where E_{i1} and E_{i2} are the vertices connected by the ith link. Below, I have provided MATLAB code that generates the edge list describing Jansen's linkage. We will be representing the linkage parameters using a struct.



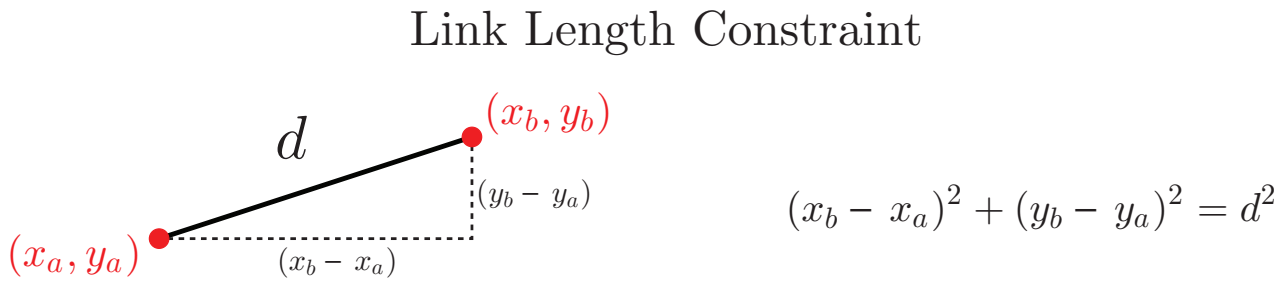

```
%initialize leg_params structure
leg_params = struct();

%number of vertices in linkage
leg_params.num_vertices = 7;

%number of links in linkage
leg_params.num_linkages = 10;

%matrix relating links to vertices
leg_params.link_to_vertex_list = ...
    [ 1, 3;... %link 1  adjacency
      3, 4;... %link 2  adjacency
      2, 3;... %link 3  adjacency
      2, 4;... %link 4  adjacency
      4, 5;... %link 5  adjacency
      2, 6;... %link 6  adjacency
      1, 6;... %link 7  adjacency
      5, 6;... %link 8  adjacency
      5, 7;... %link 9  adjacency
      6, 7 ... %link 10 adjacency
      ];
```

Linkage Constraints

Each link places a kinematic constraint on the pair of vertices it is connecting. Specifically, since the link is rigid (and therefore maintains a constant length), the distance between the pair of vertices is constant. In other words, if (x_a, y_a) and (x_b, y_b) are the coordinates of a pair of vertices connected to one another by a link of length d, then it follows that x_a, x_b, y_a, y_b, and d must satisfy:

$$(x_b - x_a)^2 + (y_b - y_a)^2 = d^2 \tag{2}$$

This relationship is visualized in the figure below:



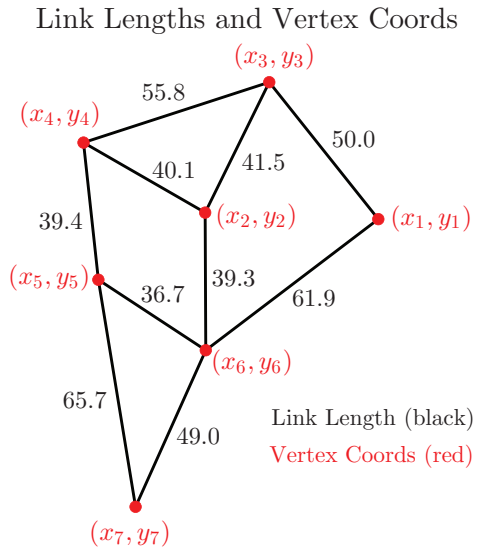
We can rewrite this equation slightly by moving the d^2 term to the other side, giving us:

$$(x_b - x_a)^2 + (y_b - y_a)^2 - d^2 = 0 \tag{3}$$

If d_i is the length of the ith link, then we can define the function $f_i(...)$ as follows:

$$f_i(x_a, y_a, x_b, y_b) = (x_b - x_a)^2 + (y_b - y_a)^2 - d_i^2 \tag{4}$$

In other words, if $[x_a, y_a, x_b, y_b]$ are the coordinates of the pair of vertices connected by the ith link in the mechanism, then $[x_a, y_a, x_b, y_b]$ is a root of $f_i(...)$! The leg linkage mechanism contains many links and many vertices, with each link corresponding to a constraint on a pair of vertices. We can represent the length of each link using a list, as shown in the MATLAB code below:




```
%list of lengths for each link
%in the leg mechanism
leg_params.link_lengths = ...
    [   50.0,... %link 1  length
        55.8,... %link 2  length
        41.5,... %link 3  length
        40.1,... %link 4  length
        39.4,... %link 5  length
        39.3,... %link 6  length
        61.9,... %link 7  length
        36.7,... %link 8  length
        65.7,... %link 9  length
        49.0 ... %link 10 length
        ];
```

With this in mind, we can construct a function that takes a vector as input and generates a vector of distance errors as output.

$$f(V) = f\left(\begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_n \\ y_n \end{bmatrix}\right) = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix}, \quad \text{where: } e_i = (x_{bi} - x_{ai})^2 + (y_{bi} - y_{ai})^2 - d_i^2, \quad V = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_n \\ y_n \end{bmatrix} \tag{5}$$

Here, V is the vector containing all of the vertex position coordinates, e_i is the distance error corresponding to the ith link, and the coordinates (x_{ai}, y_{ai}) and (x_{bi}, y_{bi}) are the positions of the two vertices that are attached to one another via the ith link. If all of the vertex coordinates satisfy the distance constraints imposed by the linkage, then $f(V)$ will return a vector of all zeros. In other words, V would be a root of f, meaning we can hand it the Newton solver to find the linkage shape. Your task is to write this function:

```
%Error function that encodes the link length constraints
%INPUTS:
%vertex_coords: a column vector containing the (x,y) coordinates of every vertex
%              in the linkage. There are two ways that I would recommend stacking
%              the coordinates. You could alternate between x and y coordinates:
%              i.e. vertex_coords = [x1;y1;x2;y2;...;xn;y_n], or alternatively
%              you could do all the x's first followed by all of the y's
%              i.e. vertex_coords = [x1;x2;...xn;y1;y2;...;yn]. You could also do
%              something else entirely, the choice is up to you.
%leg_params: a struct containing the parameters that describe the linkage
%          importantly, leg_params.link_lengths is a list of linakge lengths
%          and leg_params.link_to_vertex_list is a two column matrix where
%          leg_params.link_to_vertex_list(i,1) and
%          leg_params.link_to_vertex_list(i,2) are the pair of vertices connected
%          by the ith link in the mechanism
%OUTPUTS:
%length_errors: a column vector describing the current distance error of the ith
%               link specifically, length_errors(i) = (xb-xa)^2 + (yb-ya)^2 - d_i^2
%               where (xa,ya) and (xb,yb) are the coordinates of the vertices that
%               are connected by the ith link, and d_i is the length of the ith link
function length_errors = link_length_error_func(vertex_coords, leg_params)
    %your code here
end
```

Newton's method requires that the function being solved must take in a column vector as input and generate a column vector as output. As such, I have provided a suggested format for the input vector (vertex_coords):

$$V = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_n \\ y_n \end{bmatrix} \tag{6}$$

where we alternate between the x and y coordinates of each vertex. This is not the only way to represent these coordinates as a column vector (for instance, you could do all x values first followed by all of the y values). Regardless, the column vector format may prove unwieldy, since keeping track of the indexing for each coordinate pair is a bit tricky. For your benefit, I have provided a pair of function that will allow to convert between the column vector and a friendlier matrix form:

$$f_1(V) = M, \quad f_2(M) = V, \quad \text{where: } M = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} \tag{7}$$

The code for $f_1(V) = M$ is provided below:

```
%Converts from the column vector form of the coordinates to a
%friendlier matrix form
%INPUTS:
%coords_in = [x1;y1;x2;y2;...;xn;yn] (2n x 1 column vector)
%OUTPUTS:
%coords_out = [x1,y1;x2,y2;...;xn,yn] (n x 2 matrix)
function coords_out = column_to_matrix(coords_in)
    num_coords = length(coords_in);
    coords_out = [coords_in(1:2:(num_coords-1)),coords_in(2:2:num_coords)];
end
```

and the code for $f_2(M) = V$ is provided below:

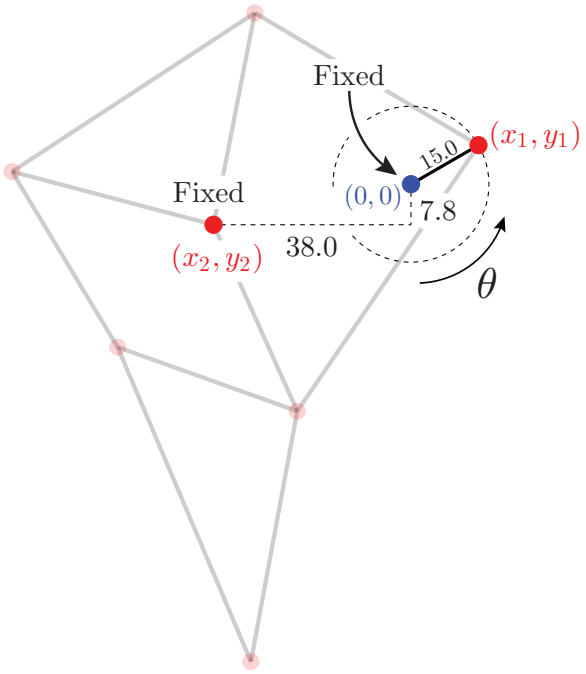
```
%Converts from the matrix form of the coordinates back to the
%original column vector form
%INPUTS:
%coords_in = [x1,y1;x2,y2;...;xn,yn] (n x 2 matrix)
%OUTPUTS:
%coords_out = [x1;y1;x2;y2;...;xn;yn] (2n x 1 column vector)
function coords_out = matrix_to_column(coords_in)
    num_coords = 2*size(coords_in,1);
    coords_out = zeros(num_coords,1);
    coords_out(1:2:(num_coords-1)) = coords_in(:,1);
    coords_out(2:2:num_coords) = coords_in(:,2);
end
```

Additional Constraints

Ideally, our constraint satisfaction problem should have the same number of constraints as variables. Each vertex uses two variables to represent (an x and a y), giving us a total of fourteen variables across seven vertices. Each link corresponds to a single constraint, giving us a total of ten constraints across ten links.

The other four constraints describe the "fixed" locations of vertex 1 and vertex 2. Specifically, vertex 1 is the tip of the crank, meaning that it travels in a circle (whose radius is the length of the crank). Furthermore, vertex 2 remains stationary relative to the center of crank circle (which we will call vertex 0). The horizontal and vertical distance between vertex 2 and vertex 0 as specified as part of the leg parameters.

Additional Constraints




```
%length of crank shaft
leg_params.crank_length = 15.0;

%fixed position coords of vertex 0
leg_params.vertex_pos0 = [0;0];

%fixed position coords of vertex 2
leg_params.vertex_pos2 = [-38.0;-7.8];
```

If $(\bar{x}_1, \bar{y}_1)$ and $(\bar{x}_2, \bar{y}_2)$ are the values that the positions of vertex 1 and vertex 2 must satisfy, then we see that (x_1, y_1) and (x_2, y_2) must satisfy the equations:

$$x_1 = \bar{x}_1, \quad y_1 = \bar{y}_1, \quad x_2 = \bar{x}_2, \quad y_2 = \bar{y}_2 \tag{8}$$

Moving all the terms to one side, we get:

$$x_1 - \bar{x}_1 = 0, \quad y_1 - \bar{y}_1 = 0, \quad x_2 - \bar{x}_2 = 0, \quad y_2 - \bar{y}_2 = 0 \tag{9}$$

From here, we can rewrite this set of equations as finding the root of a vector valued function:

$$f(\text{linkage coords}) = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix} - \begin{bmatrix} \bar{x}_1 \\ \bar{y}_1 \\ \bar{x}_2 \\ \bar{y}_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{10}$$

Where $f(...)$ is a function that takes the vector of all linkage coordinates as input. Note that the values of $[\bar{x}_1, \bar{y}_1, \bar{x}_2, \bar{y}_2]$ are determined by the leg parameters and the current angle of the crank, θ.

Your next task is to implement this function in code:

```
%Error function that encodes the fixed vertex constraints
%INPUTS:
%vertex_coords: a column vector containing the (x,y) coordinates of every vertex
%               same input as link_length_error_func
%leg_params: a struct containing the parameters that describe the linkage
%            importantly, leg_params.crank_length is the length of the crank
%            and leg_params.vertex_pos0 and leg_params.vertex_pos2 are the
%            fixed positions of the crank rotation center and vertex 2.
%theta: the current angle of the crank
%OUTPUTS:
%coord_errors:  a column vector of height four corresponding to the differences
%               between the current values of (x1,y1),(x2,y2) and
%               the fixed values that they should be
function coord_errors = fixed_coord_error_func(vertex_coords, leg_params, theta)
    %your code here
end
```

Running Newton's Method

At this point, you have written two different constraint functions that the vertex coordinates must simultaneously satisfy. We can concatenate the two constraint functions into one giant constraint by wrapping these two functions together:

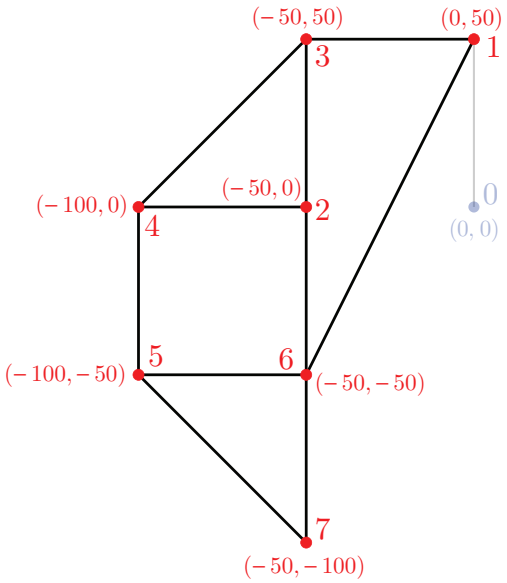
```
%Error function that encodes all necessary linkage constraints
%INPUTS:
%vertex_coords: a column vector containing the (x,y) coordinates of every vertex
%leg_params: a struct containing the parameters that describe the linkage
%theta: the current angle of the crank
%OUTPUTS:
%error_vec: a vector describing each constraint on the linkage
%           when error_vec is all zeros, the constraints are satisfied
function error_vec = linkage_error_func(vertex_coords, leg_params, theta)
    distance_errors = link_length_error_func(vertex_coords, leg_params);
    coord_errors = fixed_coord_error_func(vertex_coords, leg_params, theta);
    error_vec = [distance_errors;coord_errors];
end
```

The vertex coordinates corresponding to a legal configuration of the linkage will be a root of this function! You now have all the necessary pieces to solve for the vertex positions for a given crank angle. Write the function that performs this task:

```
%Computes the vertex coordinates that describe a legal linkage configuration
%INPUTS:
%vertex_coords_guess: a column vector containing the (x,y) coordinates of every vertex
%                     these coords are just a GUESS! It's used to seed Newton's method
%leg_params: a struct containing the parameters that describe the linkage
%theta: the desired angle of the crank
%OUTPUTS:
%vertex_coords_root: a column vector containing the (x,y) coordinates of every vertex
%                    these coords satisfy all the kinematic constraints!
function vertex_coords_root = compute_coords(vertex_coords_guess, leg_params, theta)
    %your code here
end
```

Hint!: You may need to wrap the previous linkage error function as an anonymous function so that it only takes in the vertex coordinates as input (that way you can feed it to Newton's method).

Initial Guess for Linkage Configuration




```
%column vector of initial guesses
%for each vertex location.
%in form: [x1;y1;x2;y2;...;xn;yn]
vertex_coords_guess = [...
[   0;   50];... %vertex 1 guess
[ -50;    0];... %vertex 2 guess
[ -50;   50];... %vertex 3 guess
[-100;    0];... %vertex 4 guess
[-100;  -50];... %vertex 5 guess
[ -50;  -50];... %vertex 6 guess
[ -50; -100]...  %vertex 7 guess
];
```

For better or worse, root-finding algorithms are very dependent on the initial guess provided to them. The guess doesn't need to be terrific (or accurate), but it should be reasonable. For a complex linkage, there are often many vertex configurations that satisfy the linkage distance constraints, but don't actually describe a realistic linkage configuration. Thus, we need should choose a guess that kind of looks like what the linkage shape will be. I have generated one such guess above, which you may use if you'd like. Note how this guess definitely doesn't satisfy any of the constraints, but it sort of looks like the linkage shape. It will be accurate enough for our purposes.

Visualization

Now that you are able to simulate Jansen's linkage, let's make a cool video! Create an animation that depicts the linkage moving around for several full rotations of the crank. To assist you, I have written a function that initializes a MATLAB struct that contains all the necessary plotting objects that you'd need to keep track of for the linkage. Feel free to make changes if you'd like, but at the very least, make sure to give it a read.

```
%Creates a set of plotting objects to keep track of each link drawing
%each vertex drawing, and the crank drawing
%INPUTS:
%leg_params: a struct containing the parameters that describe the linkage
%OUTPUTS:
%leg_drawing: a struct containing all the plotting objects for the linkage
%       leg_drawing.linkages is a cell array, where each element corresponds
%       to a plot of a single link (excluding the crank)
%       leg_drawing.crank is a plot of the crank link
%       leg_drawing.vertices is a cell array, where each element corresponds
%       to a plot of one of the vertices in the linkage
function leg_drawing = initialize_leg_drawing(leg_params)
    leg_drawing = struct();
    leg_drawing.linkages = cell(leg_params.num_linkages,1);

    for linkage_index = 1:leg_params.num_linkages
        leg_drawing.linkages{linkage_index} = line([0,0],[0,0],'color','k','linewidth',2);
    end

    leg_drawing.crank = line([0,0],[0,0],'color','k','linewidth',1.5);

    leg_drawing.vertices = cell(leg_params.num_vertices,1);
    for vertex_index = 1:leg_params.num_vertices
        leg_drawing.vertices{vertex_index} = line([0],[0],'marker',...
          'o','markerfacecolor','r','markeredgecolor','r','markersize',8);
    end
end
```

I have also written a template function that will update the leg plot with the current vertex coordinates. You will need to fill in some of the code on your own:

```
%Updates the plot objects that visualize the leg linkage
%for the current leg configuration
%INPUTS:
%complete_vertex_coords: a column vector containing the (x,y) coordinates of every vertex
%leg_drawing: a struct containing all the plotting objects for the linkage
%       leg_drawing.linkages is a cell array, where each element corresponds
%       to a plot of a single link (excluding the crank)
%       leg_drawing.crank is a plot of the crank link
%       leg_drawing.vertices is a cell array, where each element corresponds
%       to a plot of one of the vertices in the linkage
function update_leg_drawing(complete_vertex_coords, leg_drawing, leg_params)
    %iterate through each link, and update corresponding link plot
    for linkage_index = 1:leg_params.num_linkages

        %linkage_index is the label of the current link
        %your code here

        %line_x and line_y should both be two element arrays containing
        %the x and y coordinates of the line segment describing the current link
        line_x = %your code here
        line_y = %your code here
        set(leg_drawing.linkages{linkage_index},'xdata',line_x,'ydata',line_y);
    end

    %iterate through each vertex, and update corresponding vertex plot
    for vertex_index = 1:leg_params.num_vertices

        %vertex_index is the label of the current vertex
        %your code here

        %dot_x and dot_y should both be scalars
        %specifically the x and y coordinates of the corresponding vertex
        dot_x = %your code here
        dot_y = %your code here

        set(leg_drawing.vertices{vertex_index},'xdata',dot_x,'ydata',dot_y);
    end

    %your code here

    %crank_x and crank_y should both be two element arrays
    %containing the x and y coordinates of the line segment describing the crank
    crank_x = %your code here
    crank_y = %your code here

    set(leg_drawing.crank,'xdata',crank_x,'ydata',crank_y);
end
```

You are encouraged to use the template, but you don't have to if you don't want to. Please include an overlay of the path of the leg tip in your animation (the foot motion path from the first figure in the Strandbeest assignment).

Computing Velocity with Linear Algebra

Day 8 (Friday, September 26th) activity. Please complete before class on Tuesday, October 1st. This system presents a great opportunity to exercise our linear algebra muscles! We are going to compute the velocity of the leg tip by exploiting some properties of the Jacobian of the error function. To begin with, let's take another look at the linkage length error function, which we will call $F(V)$:

$$F(V) = \begin{bmatrix} f_1(V) \\ f_2(V) \\ \vdots \\ f_m(V) \end{bmatrix}, \quad V = \text{ column vector of vertex coordinates} \tag{11}$$

Here, $f_i(V)$ corresponds to the distance error function of the ith link, and V is the vector of all the vertex coordinates. For this exercises, I'm going to assume that V is of the form:

$$V = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \vdots \\ x_n \\ y_n \end{bmatrix} \tag{12}$$

though, it's perfectly fine if you used a different form of V (the math will basically be the same). Let's now suppose that V is a function of a single variable, which in our case is the angle of the crank, θ. In other words, each coordinate value is individually a function of θ:

$$V(\theta) = \begin{bmatrix} x_1(\theta) \\ y_1(\theta) \\ x_2(\theta) \\ y_2(\theta) \\ \vdots \\ x_n(\theta) \\ y_n(\theta) \end{bmatrix} \tag{13}$$

In this case, F can also be thought of as a function of θ, $F(V(\theta))$:

$$F(V(\theta)) = \begin{bmatrix} f_1(V(\theta)) \\ f_2(V(\theta)) \\ \vdots \\ f_m(V(\theta)) \end{bmatrix} \tag{14}$$

How can we compute $\frac{dF}{d\theta}$? Well, for any individual function f_i, we can apply the chain rule:

$$\frac{df_i}{d\theta} = \frac{\partial f_i}{\partial x_1}\frac{dx_1}{d\theta} + \frac{\partial f_i}{\partial y_1}\frac{dy_1}{d\theta} + \frac{\partial f_i}{\partial x_2}\frac{dx_2}{d\theta} + \frac{\partial f_i}{\partial y_2}\frac{dy_2}{d\theta} + +... + \frac{\partial f_i}{\partial x_n}\frac{dx_n}{d\theta} + \frac{\partial f_i}{\partial y_n}\frac{dy_n}{d\theta} \tag{15}$$

If we compile this across all values of i, we get the matrix equation:

$$\frac{dF}{d\theta} = \begin{bmatrix} \frac{df_1}{d\theta} \\ \frac{df_2}{d\theta} \\ \vdots \\ \frac{df_m}{d\theta} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial x_n} & \frac{\partial f_1}{\partial y_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial x_n} & \frac{\partial f_2}{\partial y_n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial y_1} & \frac{\partial f_m}{\partial x_2} & \frac{\partial f_m}{\partial y_2} & \cdots & \frac{\partial f_m}{\partial x_n} & \frac{\partial f_m}{\partial y_n} \end{bmatrix} \begin{bmatrix} \frac{dx_1}{d\theta} \\ \frac{dy_1}{d\theta} \\ \vdots \\ \frac{dx_n}{d\theta} \\ \frac{dy_n}{d\theta} \end{bmatrix} = J\frac{dV}{d\theta} \tag{16}$$

Where J is the Jacobian of F:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial x_n} & \frac{\partial f_1}{\partial y_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial x_n} & \frac{\partial f_2}{\partial y_n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial y_1} & \frac{\partial f_m}{\partial x_2} & \frac{\partial f_m}{\partial y_2} & \cdots & \frac{\partial f_m}{\partial x_n} & \frac{\partial f_m}{\partial y_n} \end{bmatrix} \tag{17}$$

and $\frac{dV}{d\theta}$ is the derivative of V:

$$\frac{dV}{d\theta} = \begin{bmatrix} \frac{dx_1}{d\theta} \\ \frac{dy_1}{d\theta} \\ \vdots \\ \frac{dx_n}{d\theta} \\ \frac{dy_n}{d\theta} \end{bmatrix} \tag{18}$$

This is just the derivative chain rule for vector valued functions. Since F is the linkage distance error function, we know that V is a legal configuration of vertex positions, then it is also a root of F:

$$F(V_{legal}) = 0 \tag{19}$$

Thus, if $V_l(\theta)$ corresponds to the vertex positions for each angle of the crank, we see that:

$$F(V_l(\theta)) = 0, \quad \rightarrow \quad \frac{dF}{d\theta} = \frac{d}{d\theta}(0) = 0 \tag{20}$$

Since $\frac{dF}{d\theta} = J\frac{dV}{d\theta}$, we get:

$$J\frac{dV}{d\theta} = 0 \tag{21}$$

In other words, $\frac{dV}{d\theta}$ is a member of the nullspace of J! This gives us a useful constraint on the velocity. Now, let's look at our fixed position constraints:

$$x_1 = \bar{x}_1, \quad y_1 = \bar{y}_1, \quad x_2 = \bar{x}_2, \quad y_2 = \bar{y}_2 \tag{22}$$

Remember that vertex 2 remains stationary, and vertex 1 moves along a circular path. Since the positions of vertex 1 depend on θ, we can write:

$$x_1 = \bar{x}_1(\theta), \quad y_1 = \bar{y}_1(\theta), \quad x_2 = \bar{x}_2, \quad y_2 = \bar{y}_2 \tag{23}$$

Taking the θ derivative of this expression gives us:

$$\frac{dx_1}{d\theta} = \frac{d\bar{x}_1}{d\theta}, \quad \frac{dy_1}{d\theta} = \frac{d\bar{y}_1}{d\theta}, \quad \frac{dx_2}{d\theta} = 0, \quad \frac{dy_2}{d\theta} = 0 \tag{24}$$

or in vector form:

$$\begin{bmatrix} \frac{dx_1}{d\theta} \\ \frac{dy_1}{d\theta} \\ \frac{dx_2}{d\theta} \\ \frac{dy_2}{d\theta} \end{bmatrix} = \begin{bmatrix} \frac{d\bar{x}_1}{d\theta} \\ \frac{d\bar{y}_1}{d\theta} \\ 0 \\ 0 \end{bmatrix} \tag{25}$$

Observe that the vector on the left consists of the first four elements of $\frac{dV}{d\theta}$. Thus, we can write:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 1 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 1 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 1 & 0 & \ldots \end{bmatrix} \frac{dV}{d\theta} = \begin{bmatrix} \frac{dx_1}{d\theta} \\ \frac{dy_1}{d\theta} \\ \frac{dx_2}{d\theta} \\ \frac{dy_2}{d\theta} \end{bmatrix} \tag{26}$$

or, in other words:

$$[I(4), 0]\frac{dV}{d\theta} = \begin{bmatrix} \frac{dx_1}{d\theta} \\ \frac{dy_1}{d\theta} \\ \frac{dx_2}{d\theta} \\ \frac{dy_2}{d\theta} \end{bmatrix} \tag{27}$$

Here, $I(4)$ is the 4x4 identity matrix, and 0 is a 4x10 matrix of zeros. Substituting this into our previous constraint, we get:

$$[I(4), 0]\frac{dV}{d\theta} = \begin{bmatrix} \frac{d\bar{x}_1}{d\theta} \\ \frac{d\bar{y}_1}{d\theta} \\ 0 \\ 0 \end{bmatrix} \tag{28}$$

If we stack this on top of the linkage velocity constraint, we get:

$$\begin{bmatrix} [I(4), 0] \\ J \end{bmatrix} \frac{dV}{d\theta} = M\frac{dV}{d\theta} = \begin{bmatrix} \frac{d\bar{x}_1}{d\theta} \\ \frac{d\bar{y}_1}{d\theta} \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{29}$$

We can now solve for the coordinate velocities:

$$\frac{dV}{d\theta} = M^{-1} \begin{bmatrix} \frac{d\bar{x}_1}{d\theta} \\ \frac{d\bar{y}_1}{d\theta} \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{30}$$

This derivation indicates a process for computing the velocities of various vertices in this linkage:

1. Compute the Jacobian of the linkage distance error function, J.
2. Given the current value of θ, compute the derivatives of (x_1, y_1) and (x_2, y_2):

$$\begin{bmatrix} \frac{dx_1}{d\theta} \\ \frac{dy_1}{d\theta} \\ \frac{dx_2}{d\theta} \\ \frac{dy_2}{d\theta} \end{bmatrix} = \begin{bmatrix} \frac{d\bar{x}_1}{d\theta} \\ \frac{d\bar{y}_1}{d\theta} \\ 0 \\ 0 \end{bmatrix} \tag{31}$$

3. Using these values, construct matrix M and column vector B

$$M = \begin{bmatrix} [I(4), 0] \\ J \end{bmatrix}, \quad B = \begin{bmatrix} \frac{d\bar{x}_1}{d\theta} \\ \frac{d\bar{y}_1}{d\theta} \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{32}$$

note that (for this solution technique), B should have 14 elements.

4. Compute $\frac{dV}{d\theta} = M^{-1}B$

5. (Sometimes not necessary) If θ is itself a function of time, then you can apply the chain rule to find $\frac{dV}{dt}$:

$$\frac{dV}{dt} = \frac{dV}{d\theta}\frac{d\theta}{dt} \tag{33}$$

6. Select the relevant elements of $\frac{dV}{d\theta}$ or $\frac{dV}{dt}$, and discard the remainder

Your task is to implement this as a function in MATLAB:

```
%Computes the theta derivatives of each vertex coordinate for the Jansen linkage
%INPUTS:
%vertex_coords: a column vector containing the (x,y) coordinates of every vertex
%               these are assumed to be legal values that are roots of the error funcs!
%leg_params: a struct containing the parameters that describe the linkage
%theta: the current angle of the crank
%OUTPUTS:
%dVdtheta: a column vector containing the theta derivates of each vertex coord
function dVdtheta = compute_velocities(vertex_coords, leg_params, theta)
    %your code here
end
```

Using this function, you should be able to compute the velocity of the leg tip at any given point in time. Include the velocity of the leg tip as an overlay on your animation. This should be an arrow that starts at the leg tip, and points in the direction of motion. This vector should be tangent to the curve describing the leg tip path (that you are also supposed to include as an overlay, see the previous section). You may want to scale the vector in the visualization by a constant factor so that it's not too long/short.

To further verify that your function is working properly, we will numerically compute the velocity of the leg tip in a different way. You have already written a function that generates the vertex positions for any given crank angle:

```
function vertex_coords_root = compute_coords(vertex_coords_guess, leg_params, theta)
```

Use this function and your Jacobian approximation function to compute $\frac{dV}{d\theta}$. You may need to wrap the compute coords function as an anonymous function in order to make it a function of just θ (and not any other variables). Once you have completed this, generate plots that compare your two different computed values of $\frac{dV}{d\theta}$. Specifically, I want to see two plots:

1. A plot of $\frac{dx_{tip}}{d\theta}$ for $\theta \in [0, 2\pi]$ for both methods of computing the derivative.

2. A plot of $\frac{dy_{tip}}{d\theta}$ for $\theta \in [0, 2\pi]$ for both methods of computing the derivative.

I would recommend using a different color for the two different computation methods. I would also recommend plotting one method using a solid line and the other method with a dashed line (the dashed line should be the one on top), so that both versions are visible, even if their values are extremely similar.

Deliverables

Day 9 (Tuesday, September 30th) will be dedicated to wrapping up any loose ends and writing your lab reports. Please make sure to submit these deliverables by 11:59 PM on Thursday, October 2nd.

Your primary deliverable will be in the form of a lab report that documents what you have accomplished and learned. How you write the lab report is up to you, but there are a few items that you should make sure to include, and a few submission guidelines. guidelines.

- Remember that each team will be submitting a single assignment. You will need to add yourself to your team's 'assignment02' group on Canvas. Make sure to include the names of you and your teammates at the top of your lab report
- It is expected that each team-member will contribute equally to the lab report itself. You are strongly discouraged from having one teammate do all the coding and one teammate writing the entire lab report.
- The lab report should be saved as a pdf.
- At the start of the lab report, please approximate how much time each teammate spent working on the assignment (you won't be judged on time spent, but I need it to gauge the assignment difficulty).
- Include a description in your own words of the process that you used to simulate the Strandbeest linkage.
- If you had to start this mini-project over from scratch, what would you do the same? What would you do differently?
- Write a short reflection describing three things that you learned while working on this project.
- Please include plots comparing the values of $\frac{dx_{tip}}{d\theta}$ and $\frac{dy_{tip}}{d\theta}$ for your two different methods of computation.
- Please submit the code that you have written over the course of this module. This code should be readable and adequately commented. Code should be submitted as several '.m' files (livescripts will not be accepted).
- Please upload your Strandbeest leg to youtube, and then include the link to the video as a submission on Canvas. This submission should be separate from the lab report (I would prefer not to have to hunt around each lab report looking for a youtube link when grading). Remember the following:
  - The animation should show the crank turn for multiple cycles.
  - Include an overlay (or underlay) of the path of the leg tip for one full cycle (see the first figure in the Strandbeest assignment to see what that should look like).
  - Include an overlay of the leg tip velocity vector.