Lab: Position Velocity Acceleration (PVA) Analysis

PVA Lab - Background Information

In lecture, you learned how to do PVA analysis by writing vector loop equations, separating the imaginary and real components then analytically solving the resulting system of equations. While this method gives exact solutions, it becomes more difficult to apply when you have more complex systems with many vector loops and simultaneous equations to solve. For these types of systems, we will show you how to use computational methods to do the PVA analysis. Although these methods cannot give you exact answers like the vector loop method, they can be tuned to give results to any degree of accuracy needed.

Consider the four-bar linkage of Figure 1 where the ground link is link 4 and the input crank is link 3. The initial linkage position is given by solid lines.

Note that if you know the initial coordinates of each joint at time t_0, and how they are connected to the links, all information about the four-bar linkage can be calculated. From these initial conditions, the initial link lengths L_1, L_2, L_3, and L_4 can be calculated by the code and must remain the same no matter how the node positions change.

Figure 1 demonstrates the labeling scheme used to implement this computational procedure. The initial nodes or joints of the mechanism are labeled by the circled numbers.

The code uses the complex plane to describe the geometry of the linkage where the horizontal axis is the real part, and the vertical axis is the imaginary part. A joint (e.g. 2) is fully described in this plane by its complex value (e.g. X_2). The solid lines show the link geometries for the starting time, t_0, where the links are arbitrarily labeled by the numbers enclosed in squares.

Other inputs to the code include:

- which link is the ground link (always fixed);
- which link is the crank
- crank angular velocity

x_1	0
x_2	1+5i
x_3	2+i
x_4	3

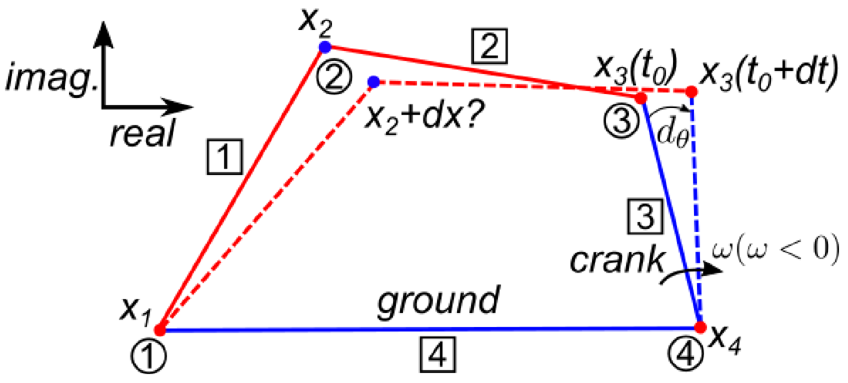


Figure 1: PVA labeling procedure for four bar linkage code

The first step of the code is to incrementally rotate the input crank (link 3) by a small amount to a new position denoted by the blue dotted line. The code registers that link 4 remains fixed during the simulation, acting as the ground link. Since we know the input angular velocity of the crank and the starting positions, the positions of the crank/ground link nodes (x_1, x_3, x_4) will be known exactly for every time step in the simulation using simple explicit kinematic equations. Next the code needs to compute x_2 as a function of time given that the link lengths are fixed.

<u>How does the code numerically compute the kinematics of joint 2?</u>

When the crank node is first brought to a slightly perturbed position as shown above, the joint which does not have a prescribed position (the one that is not part of the ground or crank links), as shown by the blue dot joint 2, needs to be moved to a position, x_2+dx, such that the initial link lengths (L_1, L_2, L_3, and L_4) are preserved. Put another way, when x_3 is moved to $x_3(t_0+dt)$, x_2 needs to be slightly adjusted to x_2+dx such that the link lengths remain the same as they were initially. Since we do not know what dx is, we need a function to represent how far the red link lengths are from what they should be for an arbitrary perturbation, dx, added to x_2. For this system, the objective function would look like,

$$f(dx) = \begin{bmatrix} |x_3(t_0+dt) - x_2(t_0+dt)| - L_2 \\ |x_2(t_0+dt) - x_1| - L_1 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

where |...| denotes the L_2 vector norm, or the length of link 2. The perturbation, dx, that preserves link lengths in the system is the one in which [f1; f2] = [0; 0]. Therefore, we only need to solve f(dx)=0 numerically in Python. In doing so, the 2 unknowns to the function, dx, (which has 1 real and 1 imaginary component) need to be found to solve both f1 = 0, f2 = 0. In Python, the *scipy.optimize.least_squares* function, a nonlinear equation solver, can be used to solve this equation. By supplying an input guess, such as dx=0+0i for this system, the nonlinear equation solver systematically improves dx based on the derivative (Jacobian) of the function until it finds a solution. The details of this function are beyond the scope of this course. However, if you are interested, you can more about such methods in the textbook (Section 4.14).

The code is structured as follows:

<u>Section 1: Input</u>

To establish the link lengths in the system, we use two inputs, *initial_node_positions* and *connectivity_matrix.* ***initial_node_positions*** is a single column vector containing the initial joint (node) positions or coordinates in complex form, the order of the nodes in the array determines the node index. For the above four bar linkage, it would take the form,

$$initial\_node\_positions = \begin{bmatrix} x_1 + y_1 j \\ x_2 + y_2 j \\ x_3 + y_3 j \\ x_4 + y_4 j \end{bmatrix}$$

connectivity_matrix is the connectivity matrix which specifies how the nodes/joints in the mechanism are connected. It is a two-column array where the *i*th row gives the two node indices that make link *i.*

Alternatively, the row represents the link index, while the columns represent the two node indices that the link is comprised of. For example, *connectivity_matrix* for the above mechanism would be,

$$connectivty\_matrix = \begin{bmatrix} [0,1] \\ [1,2] \\ [2,3] \\ [3,0] \end{bmatrix} \begin{matrix} (node\ 1, node\ 2)(link\ 1) \\ (node\ 2, node\ 3)(link\ 2) \\ (node\ 3, node\ 4)(link\ 3) \\ (node\ 4, node\ 1)(link\ 4) \end{matrix}$$

NOTE: Python will use this information to calculate the link lengths, so the node indices need to start from 0 and not 1.

Now we need to specify the link indices corresponding to the input crank and ground links.

ground_links_idx is a list of the indices of the ground links in the connectivity matrix (for the above example, since Python starts counting at 0, so this corresponds to *ground_links_idx* = [3]). Multiple ground links can be specified for redundant mechanisms or for mechanisms with slider constraints.

crank_link_idx is the link index in the connectivity matrix of the crank link input (2 for the above example, since Python starts counting at 0).

motor_node_idx is the node that the motor is at (3 for the above example, since Python starts counting at 0). This node must be shared by the crank link otherwise you will get errors.

There are a number of optional inputs available to analyze more complex mechanisms than the above example. These parameters may be especially useful for your final project PVA/DFA analysis. See the **Appendix** at the end of this document for more details.

Next, we must specify the total amount of time to run the simulation- ***tperiod*** *(in seconds),* e.g., 36 seconds; the incremental amount of time or time step that when moving the system- ***dt*** *(in seconds),* e.g., 0.1 seconds*; a*nd the constant angular velocity that the input crank should travel- ***crank_angular_velocity*** *(deg/sec)* assuming a positive right hand rule convention for rotation shown in the above figure (e.g. 10 deg/sec).

The rest of the code is structured as follows. Find the section headers in the code and read the accompanying code comments to better understand what each line is doing.
Section 2: Makes useful book-keeping information about the system.
Section 3: Allocation/Initialization of time varying node positions *xnode.*
Section 4: Puts in initial fixed ground link node positions at each time step in corresponding position array since they are not changing with time.
Section 6: Computes link lengths from initial node positions.
Section 7: Loop over each time step and find adjustments, dx, to preserve link lengths. This uses the objective *link_length_difference* function to find dx as summarized previously.
Section 8: Animates Linkage
Section 9: Computes the node velocities and accelerations from node positions.

<u>PVA Lab – Pre-Lab Exercise</u>

Read this manual and complete the Pre-lab assignment posted on Gradescope before the respective due dates. Students are **only allowed one attempt**, and all of the answers can be found in this lab document.

PVA Lab – In-lab Exercises

Before starting:

1. *All code templates are available at the link:*

https://drive.google.com/drive/folders/1IC0mClk4FPjRWD7umifWuxMZhBvD2g9T?usp=sharing

2. *Make a local copy of the ME370_PVA_in_lab.ipynb notebook to your Google colaboratory notebook (the file constitutes 3 blocks/cells of code). To make a local copy, download the notebook from the link above, open Google Colaboratory in your browser, go to File->Upload notebook (you may need to sign-in).* ***You should not modify the first cell.***
3. *Note: This is an* ***individual*** *lab, and you need to fill out the gradescope questions individually.*

Question 0

a) Run the first code cell.

Question 1

In the second code cell, scroll down and fill in the rest of Section 6 to compute the link lengths from the initial node positions. You need to loop over each row (link) in the ***connectivity_matrix,*** to extract the node indices that make each link. Then, use these node indices to extract the node positions from ***initial_node_positions.*** With these positions (which are complex values), compute the link length and store the value in a numpy array named ***link_lengths.***

Question 2

a) **In the second code cell,** scroll down and complete Section 9 of the code to compute the velocities of the node points in the simulation (see Hint[1] below). The code to calculate the acceleration of the node positions has already been provided for you.

b) Make a plot of the output rocker node's velocity and acceleration (In X direction) vs crank angle. To do this, you will need to add code to determine the crank angle at each time step (in the same section 9 below your code to part a). Use the values in the first column of the array ***node_positions*** (or ***initial_node_positions***) to help you find the initial crank angle, then use your values for ***tperiod, dt,*** and ***crank_angular_velocity*** to create a vector of the crank angle at each point in time.

c) Run the second code cell.

Question 3

a) **In the third code cell,** fill in the input parameter ***initial_node_positions, connectivity_matrix, ground_link_idx, crank_link_idx, motor_node_idx, dt, crank_angular_velocity*** from Section 1 (as described in the manual above) to model a crank-rocker system. You should use the node positions provided in Fig. 1 from PVA background reading. The angular velocity can be assumed to be clockwise.

b) Run the third code cell for ***tperiod*** = 5 to see if your code is working. If there are no errors, then change it to the value from Section 1 above. This cell may take a couple of minutes to run, so please be patient (you can see the execution time in bottom-right of your window).

Once completed, run the generated animation by clicking on the play button (you can control the speed etc. of the animation using other indicated buttons beside the play button). Include all the figures in your post-lab report along with your code for each question above; the animation is not required to be included. Additionally, copy the following AI statement to your post-lab report and select the appropriate option:

Select one of the following options:
a) My answer was created by a Gen AI algorithm, and I have not modified it
b) My answer was created by a Gen AI algorithm, and I have made some minor changes.
c) My answer was created by a Gen AI algorithm, and I have made major changes.
d) My answer was created solely by myself.
e) If I used Gen AI, I used ___ (name of program).

[1]*Hint:* $\mathrm{v} = \frac{\mathrm{x(t+dt)-x(t)}}{\mathrm{dt}}$ *or* $\frac{x(t+dt)-x(t-dt)}{2dt}$ *and* $\mathrm{a} = \frac{\mathrm{v(t+dt)-v(t)}}{\mathrm{dt}}$ *or* $\frac{v(t+dt)-v(t-dt)}{2dt}$.

PVA Post Lab assignment (60 points)

The goal of this exercise is for you to understand how the PVA python program operates for more complex mechanisms. You are encouraged to make use of this code for the project.

Before starting:

1. *All code templates are available at the link:*

https://drive.google.com/drive/folders/1IC0mClk4FPjRWD7umifWuxMZhBvD2g9T?usp=sharing

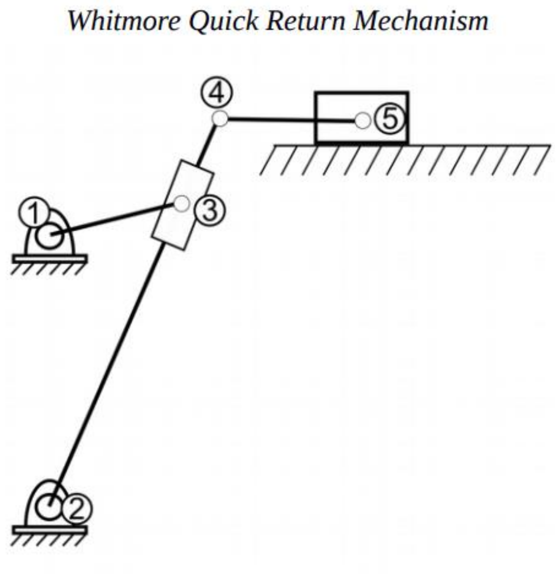
2. *Make a local copy of the ME370_PVA_Postlab.ipynb file to you Google colaboratory notebook, which constitutes 3 cells of code. To make a local copy, download the notebook from the link above, open Google Colaboratory in your browser, go to File->Upload notebook (you may need to sign-in).* ***You should not modify the first code cell.***
3. *Note: This is an* ***individual*** *lab, and you need to fill out the gradescope questions individually.*

Question 4

a) Copy the code that you wrote for in-lab Question 1 and paste it into the corresponding Section 6 of second code cell of the *ME370_PVA_Postlab.ipynb* notebook.
b) Copy all the code that you wrote for in-lab Question 2 and paste it into the corresponding Section 9 of second code cell of the *ME370_PVA_Postlab.ipynb* notebook.
c) **In the third code cell**, model the linkage system, as shown below. You will need to use the information provided in the **Appendix** below on slider constraints. NOTE: Python indices start from 0 and not 1.
d) Adjust the crank velocity such that it completes **1 revolution in 9 seconds (counter-clockwise).**

A Whitmore quick return mechanism is shown in Fig. 1 below. It consists of 6 links (including ground), 5 rotary joints (as numbered in the picture) and 2 sliding constraints. The distance between nodes 1 and 2 is 5 meters and they are vertically aligned. The ratio of the distances between nodes 2 and 3 to that of the distance between 3 and 4 is 3:1. Nodes 4 and 5 are initially horizontally aligned and 3 meters apart. Finally, node 4 is 5 meters away (total) from node 1 and the ground with the sliding block is 3 meters above node 1. You may assume the slider friction to be zero.

Lab: Position Velocity Acceleration (PVA) Analysis



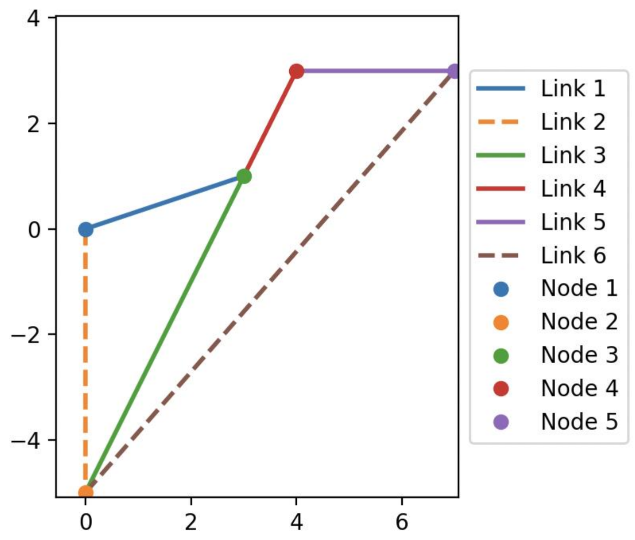


Figure 1: Whitmore quick return mechanism (left) and the Python implementation (right)

Tips

- *It may be beneficial for you to model a second ground link between nodes 2 and 5.*
- *To ensure that slider 3 functions as intended, you will have to specify a fixed angle constraint for the links on either side of it*
- *Remember to look at the solutions posted after the PVA in-lab activities before doing the post-lab to ensure that your PVA.py code works correctly.*

Deliverables

1. Plot the position, velocity and acceleration of the horizontal slider (node 5) as a function of time for **4 full cycles** of the crank (link connecting nodes 1 to 3) by running the code cells one to three in sequence. Adjust the crank velocity such that it completes **1 revolution in 9 seconds (counter-clockwise)**.
2. Include the code you modified at the end of your PDF submission. **Comment** on the modifications and additions you had to make to the code to model this linkage.
3. Copy the following AI statement and include in the postlab with the appropriately selected option:

   Select one of the following options:
   a) My answer was created by a Gen AI algorithm, and I have not modified it
   b) My answer was created by a Gen AI algorithm, and I have made some minor changes.
   c) My answer was created by a Gen AI algorithm, and I have made major changes.
   d) My answer was created solely by myself.
   e) If I used Gen AI, I used ___ (name of program).

Appendix – Setting slider constraints

Additional Section 1 Inputs:

sliders allows you to specify slider constraints by assigning an array for each slider. Example: **sliders = [[2,2,3],[4,5,1,0]]**.

The array entry for a slider constraint takes the form:

[node_index, link_index, x_direction, y_direction]

Node index is the index of the node you want to be able to slide. Link index is the link index that you want the node to slide in relation to. You can equivalently think of this link as the one whose length can change during the simulation. For this reason, maintaining its length during the simulation is not required so it is omitted from the objective function discussed before. For most mechanisms, the link index will likely correspond to the ground link index. x_direction is the x component of a unit vector denoting the direction of constrained sliding motion from the initial position of the corresponding sliding node. Likewise, y_direction is the y component of this unit vector. So, for example, if you want node 3 to slide in relation to the ground link with link 4 in the horizontal direction, you will specify **sliders = [[2,3,1,0],]. (remember that indices start at 0 not 1)**

In this case, a slider of this construction has fixed sliding motion relative to ground during the entire simulation so we will refer to this as a 'fixed slider constraint'. Another slider constraint that has been included to this code is a 'moving slider constraint.' For this type of slider, a node has constrained sliding motion relative to two moving linkages that can vary length and orientation during the simulation. A 'moving slider constraint' can be viewed in the Whitmore quick return mechanism shown below. This constraint is the same as a slotted pin joint or half joint. The equivalent python mechanism according to the new code format is given on the right.

Node 3 is free to move along the changing length of Link 3 and 4. To specify this kind of constraint in the code, an array entry is made to the slider cell variable with only 3 components such as, **sliders=[[Node index, Link 1 index, Link 2 index],]**. Node index is the node that is free to slide. Link 1 index is one link index that the node can slide in relation to. Link 2 index is the second link the node can slide in relation to. The slider entry for this mechanism would be **sliders =[[3,3,4],]** There are two things to consider when using this kind of constraint. First, the initial node configuration that makes up the two slider links must be collinear. They must be constructed on the same line or you will likely get errors during the simulation. Second, you will likely need to constrain the angle of the two links during the entire simulation (constrain them to remain collinear). You can use a 'fixed angle constraint' to do so.

If you have no slider constraints in your mechanism, you must still specify an empty cell for the variable sliders (ex: **sliders = []).**

- ***links_with_fixed_angle*** holds an angle between two links fixed during the simulation. The input is an array where each row specifies two links to maintain a fixed angle between. Both links must share a node. If no such constraints are present, you must specify an empty array (ex: **links_with_fixed_angle = []**).

- ***rotation_fixed_nodes*** fixes a node so that links connecting at that point cannot rotate. It is similar to the fixed angle constraint but easier to apply for collective link angle fixing. A row entry can be added to the ***rotation_fixed_nodes*** array denoting the node whose connecting links cannot rotate. If there is no constraint on node rotations, you must still specify an empty array (ex: **rotation_fixed_nodes = []**).

Supplementary Information about Sliders in this Code

In order to incorporate sliding constraints, the variables ***sliders*** and ***links_with_fixed_angle*** have to be correctly defined. There are two types of sliding constraints and thus two forms of declaring the variable slider. The first form is used if the sliding node is constrained to the ground, such as the node 5 in Fig. 1 above. In this case, the format is sliders = [[node_index, link_index,x_direction,y_direction]].

The meaning of each of these variables inside the brackets can be found in the previous section.

The second type of sliding constraint is used if the node is sliding in a link that is moving with the linkage, such as node 3 in the figure. For the second case, you should artificially introduce two links (3 and 4 in the figure), whose length can change over time and then define a sliding constraint as [node index, link_index1, link_index2] or simply [2,2,3] using the nomenclature shown on Fig. 1. Note that this computational trick is taken into account by the function ***link_length_difference***.

The code, however, does not automatically know that links 3 and 4 are physically the same, so you need to introduce a rigidity constraint into the system. That is done by fixing the angle between these two links through the use of variable ***links_with_fixed_angle*** as **links_with_fixed_angle= [[link1,link2]]** where link1 and link2 are the link indices that are rotationally coupled together.

In the case of multiple sliding constraints, the variable ***sliders*** should be of the form **sliders = [[constraint1], [constraint2], etc.]** where each term inside a bracket is in one of the forms previously defined.

Lab: Position Velocity Acceleration (PVA) Analysis

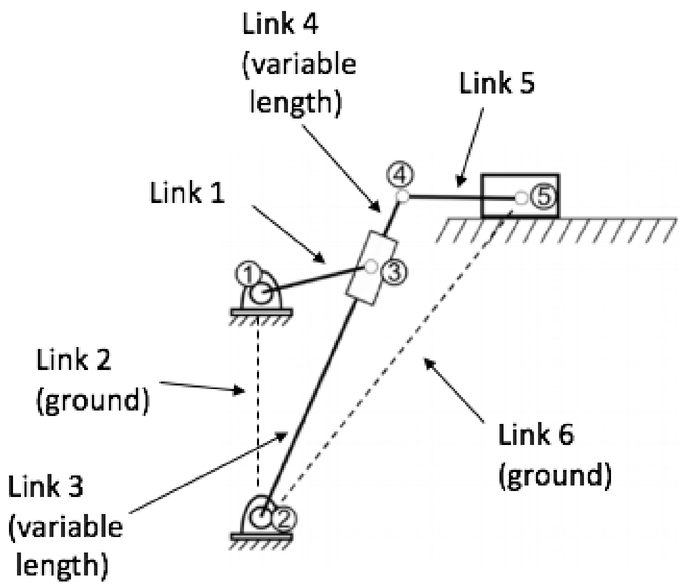


Figure 2