Computational Fabrication Spring 2020

Programming Assignment 3: Computational Linkage Design

A linkage is an assembly of bodies connected to manage forces and movement. In this assignment you will design some cool linkages, which will be physically manufactured in a later assignment. We will be only looking at kinematics and ignore forces, and furthermore, we will limit ourselves to 2D planar structures with revolute joints. Provided with this assignment is a fully functional Matlab code for simulating linkages.

In this assignment you will be responsible for:

1. Modifying the code to allow oscillatory driver input (instead of rotational)
2. Designing several linkages

The remainder of this document is organized as follows:

1. Getting Started
2. Starter Code and Implementation Notes
3. Your Tasks
4. Extra Credit
5. Submission Instructions

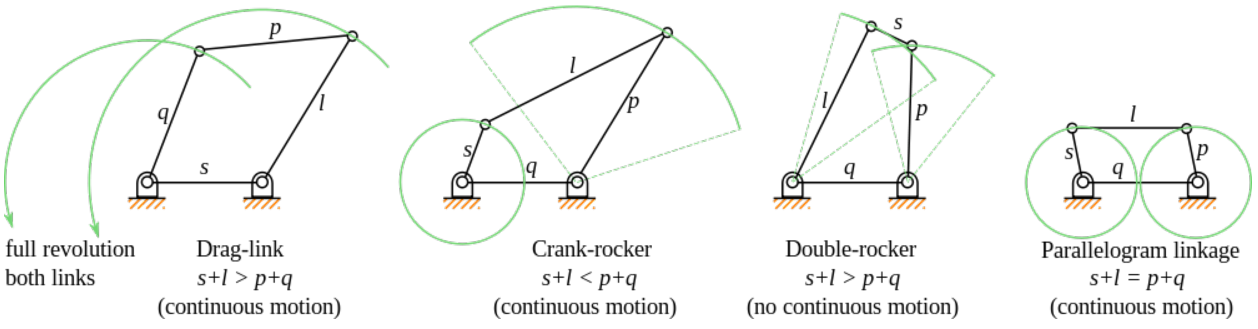

Figure 1: Four-bar linkages [wikipedia]. s = smallest link length; l = longest link length; p, q = other two lengths.

1 Getting Started

1.1 Running the Code

The provided code is standalone and does not require any external libraries. Here we outline how to run the code from Matlab's command window.

1. Extract `linkage.m` to `<WORKINGDIR>`.
2. Open Matlab and go to the command window.
3. `>> cd <WORKINGDIR>`
4. `>> linkage(0)`

Matlab will then open a figure window that shows the simulation of the sample linkage. The pin constraint between the right and top links is randomized, so if you run the code multiple times, you will get a slightly different mechanism each time. The argument "`0`" specifies the scene to run, which in this case is a crank-rocker mechanism. Each new linkage mechanism that you create should have a new scene number.

2 Starter Code and Implementation Notes

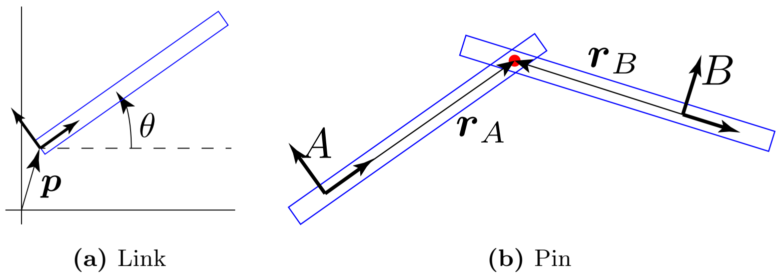

Figure 2: (a) A link defined by its orientation, θ, and position, $\boldsymbol{p}$. (b) A pin between two links. $\boldsymbol{r}_A$ and $\boldsymbol{r}_B$ define where the pin location is with respect to the links.

2.1 How to Create Scenes

To create a scene, we need a list of links and a list of constraints between the links. This is done inside the `switch` statement at the top of the code. The sample four-bar linkage scene is defined in `case 0`.

Link: A link is modeled as a rigid body, and in 2D, it has three degrees of freedom: $\theta \in \mathbb{R}$ and $\boldsymbol{p} \in \mathbb{R}^2$ (Fig. 2a). θ specifies the orientation of the link with respect to the positive x-axis, and $\boldsymbol{p}$ specifies the position of the center of rotation. Given a point, $\boldsymbol{r}$, in the link's local coordinates, the world coordinates of that point can be computed as

$$\boldsymbol{x} = R(\theta)\,\boldsymbol{r} + \boldsymbol{p}, \tag{1}$$

where R is the rotation matrix given by

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}. \tag{2}$$

For example, the following code creates a horizontal link at the origin.

```
links(1).angle = 0;
links(1).pos = [0;0];
links(1).verts = [0,-0.1;2,-0.1;2,0.1;0,0.1]';
```

The kinematics of a link is fully expressed using θ and $\boldsymbol{p}$. In order to draw it on screen, however, we need to give it a mesh. In this assignment, we simply assign to each link a list of vertices expressed in the link's local coordinates and transform these vertices into world coordinates whenever we move the link. Note that the mesh is for display only—the simulation would work just fine even if we do not have a mesh. In the example above, the vertices are

$$\begin{pmatrix} 0 \\ -0.1 \end{pmatrix}, \quad \begin{pmatrix} 2 \\ -0.1 \end{pmatrix}, \quad \begin{pmatrix} 2 \\ 0.1 \end{pmatrix}, \quad \begin{pmatrix} 0 \\ 0.1 \end{pmatrix},$$

which defines a rectangle with the link's coordinate frame defined at the left end, as in Fig. 2a. These vectors are stored column wise in the `links(1).verts` matrix.

Additionally, we specify that one of the links is grounded, meaning that it cannot move, and another link is the driver, meaning that its orientation and position are specified procedurally. For example, `grounded = 1; driver = 2;` specifies that `links(1)` is grounded, and `links(2)` is the driver.

Pin: A pin constrains two links with a revolute joint. It stores the indices of the two links to constrain (say A and B) as well as where on the two links the pin constraint is located: $\boldsymbol{r}_A$ and $\boldsymbol{r}_B$ (Fig. 2b). The constraint should then make sure that, if $\boldsymbol{r}_A$ and $\boldsymbol{r}_B$ are transformed to world space (using Eq. 1), their locations match. (If you are curious how this is actually implemented, see Sec. 2.2.)

The following code creates a pin between `links(1)` and `links(2)` as shown in Fig. 2b.

```
pins(1).links = [1,2];
pins(1).pts = [4,0;-4,0]';
```

The `links` field specifies that the pin is between links 1 and 2, and the `pts` field specifies that

$$\boldsymbol{r}_A = \begin{pmatrix} 4 \\ 0 \end{pmatrix}, \quad \boldsymbol{r}_B = \begin{pmatrix} -4 \\ 0 \end{pmatrix}.$$

Particle: To accentuate the interesting motions of linkages, we can add tracer particles to the links (like the green lines in Fig. 1.) The following code creates a particle.

```
particles(1).link = 4;
particles(1).pt = [0.5;0.1];
```

The `link` field specifies the parent link, and the `pt` field specifies where on this parent link the particle is located (expressed in the parent link's coordinates).

Hint: When creating scenes, the links do not need to be precisely positioned, since they will snap to configurations that satisfy all the constraints that you specify, as long as those constraints are satisfiable.

2.2 How the Simulation Works (optional reading)

After the scene creation process finishes, we have a list of links, pins, and particles. The simulator then takes this information and does the following:

```
while simulating
    1. Procedurally set the driver angle
    2. Solve for linkage orientations and positions
    3. Update particle positions
    4. Draw scene
end
```

In Step 1, we manually specify the target angle of the driver link. As long as the mechanism has a single degree of freedom, the orientations and the positions of the rest of links can be solved for in Step 2 using nonlinear least squares. In Step 3, we compute the world positions of the particles given the newly updated link configurations. Finally, in Step 4, we draw the scene on the screen.

The meat of the simulator is in Step 2. The optimization variables in the nonlinear least squares problem are the orientations, θ, and positions, $\boldsymbol{p}$, of all the links, including grounded and driver links. We will use $q_i = [\theta_i, \boldsymbol{p}_i]^T$ to denote the configuration of the i^{th} link. We are looking for $q_1, \ldots, q_n$ so that all of the constraints are satisfied. There are two types of constraints implemented so far: prescribed and pin. In both cases, we'll express the constraint in the form $c(q) = 0$.

Prescribed: Grounded and driver links have their configurations manually specified. We can express this as a constraint on the orientation θ and the position $\boldsymbol{p}$ of the link.

$$c_\theta(q) = \theta - \theta_{\text{target}}, \quad c_p(q) = \boldsymbol{p} - \boldsymbol{p}_{\text{target}}. \tag{3}$$

For grounded links, the target orientation and position are fixed throughout the simulation, whereas for driver links, they are specified procedurally.

Pin: A pin constrains two links so that they share a common point. Let the two links be denoted by A and B. We can express the constraint as

$$\begin{aligned} c_{\text{pin}}(q; \boldsymbol{r}) &= \boldsymbol{x}_A - \boldsymbol{x}_B \\ &= (R(\theta_A)\,\boldsymbol{r}_A + \boldsymbol{p}_A) - (R(\theta_B)\,\boldsymbol{r}_B + \boldsymbol{p}_B). \end{aligned} \tag{4}$$

We are looking for orientations and positions of the two links such that this constraint function evaluates to zero. The orientations and positions of the two links, θ and $\boldsymbol{p}$, are the variables of this constraint function, whereas the locations of the pin joints expressed in local coordinates, $\boldsymbol{r}_A$ and $\boldsymbol{r}_B$, are the parameters of this constraint function.

Let c be the concatenation of all the constraints. We are looking for the orientations and positions of all the joints (θ_i and $\boldsymbol{p}_i$) such that the constraint violations are minimized:

$$\underset{q}{\text{minimize}} \quad \frac{1}{2} c^T c. \tag{5}$$

We can solve this easily in Matlab using the function `lsqnonlin`. What we need to provide is a function that evaluates the vector-valued function c and its Jacobian, $J = \partial c / \partial q$. For grounded and driver constraints, the Jacobian is just the identity: $J_\theta = 1, J_p = I$. For pin constraints, the Jacobian is

$$\begin{aligned} J &= \begin{pmatrix} \partial c/\partial \theta_A & \partial c/\partial \boldsymbol{p}_A & \partial c/\partial \theta_B & \partial c/\partial \boldsymbol{p}_B \end{pmatrix} \\ &= \begin{pmatrix} R'(\theta_A)\, \boldsymbol{r}_A & I & -R'(\theta_B)\, \boldsymbol{r}_B & -I \end{pmatrix} \end{aligned} \tag{6}$$

where

$$R'(\theta) = \begin{pmatrix} -\sin\theta & -\cos\theta \\ \cos\theta & -\sin\theta \end{pmatrix}. \tag{7}$$

3 Your Tasks

1. The first task is to modify the driver so that it can also generate oscillatory motion in addition to rotational motion. Currently, the piece of code that produces rotational motion is

```
...
dt = 0.01;
angVel = 2*pi;
while t < T
  links(driver).angleTarget = links(driver).angleTarget + dt*angVel;
  ...
end
```

   This integrates the target angle with a constant angular velocity (2π radians per second), so that the driver rotates at a constant rate. Your task here is to modify this code so that you can specify a range that the driver oscillates between. For example, if the range is $[-\pi/3, \pi]$, then the driver should go back and forth between $-\pi/3$ and π at some set frequency. You may not need `angVel` any more, since the angular velocity will no longer be constant.

2. Design a family of four-bar linkages [`http://en.wikipedia.org/wiki/Four-bar_linkage`]. See also Fig. 1. The sample code (`linkage(0)`) produces a crank-rocker mechanism, since the shortest link, s, is the crank (driver). Copy and paste this sample code and create a drag-link mechanism and a double-rocker mechanism. Place all the code in the appropriate `case` block. For the double-rocker, the driver motion must be oscillatory, not rotational.

3. Design a Hoekens linkage [`http://en.wikipedia.org/wiki/Hoekens_linkage`]. This is a crank-rocker mechanism that produces a nearly straight line. Make sure you add a tracer particle to show the trajectory of the tip.

4. Design a Peaucellier-Lipkin linkage [`http://en.wikipedia.org/wiki/Peaucellier-Lipkin_linkage`]. "This was the first planar linkage capable of transforming rotary motion into perfect straight-line motion, and vice versa." Again, make sure you add a tracer particle to show the trajectory of the tip.

5. Design a Klann linkage [`http://en.wikipedia.org/wiki/Klann_linkage`] [`http://www.mekanizmalar.com/mechanicalspider.html`]. For this mechanism, the geometry of some of the links will no longer be rectangles. You will need to modify the `verts` field of `link` to make this change. Note that the `drawScene()` function draws the link geometry by connecting the vertices in the order they are defined.

4 Extra Credit

The current code only allows for pin joints. For extra credit, implement a sliding joint and design a scissor mechanism [`http://en.wikipedia.org/wiki/Scissor_mechanism`]. You will have to modify, among other things, `objFun()`. This function is called by `lsqnonlin` to evaluate the constraints. To make this modification a little easier, you may want to ask `lsqnonlin` to use finite differencing instead of using analytical gradients. You can do this by setting `'Jacobian'` from `'on'` to `'off'` in `optimset` (or `optimoptions`). The code will be slower with finite differencing, but providing analytical gradients can be tricky business.

5 Submission Instructions

Please provide a report with your submission (PDF). The report should include the following:

- Images of all mechanisms that you designed.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we are much more likely to assign partial credit if you help us understand what is going on.
- Did you do any of the extra credit? Include an image of the mechanism. If there was a substantial amount of work involved, describe what and how you did it.
- Got any comments about this assignment that you would like to share?

Remember, these assignments are to be done on your own. Please do not share code or implementation details with other students.

Submit your assignment on Courseville by the due date. Please submit a single archive (`.zip` or `.tar.gz`) containing:

- Your Matlab code (**which should run simply by calling `linkage(n)`, where `n=0,1,...`**).
- Any additional files that are necessary.
- The `PDF` file.