Assignment 3: Simple Sota Kinematics

Drawing from your experience in Assignment 1, in this assignment you will engage with a similar exercise: we need to work through the code and math to have flexible high-level control of the robot's actuators in terms of what the outcomes in the world are, not each motor's specific settings.

The main goals of this assignment are:

- Gain comfort working with a distributed build environment.
- Practice employing the perspective taking and frame translation material learned in class
- Gain a deep understanding of the math behind linear-transform forward kinematics using linear algebra
- Gain an understanding of using Jacobians to solve inverse kinematics, including how to employ numerical optimization.

Build Environment

The build environment is a little more complex than last time. Spend time early to make sure you can get code to compile, upload, and run on the Sota robot, before you start this assignment. Check the online primer and provided sample project.

Hints and helper functions

Several sample programs were included that provide examples of how to program the robot, and some tips about how to print live data and work with the matrix library.

There is a lot going on in this assignment. Don't worry about optimization and efficiency until you need it. Get it working first, then re-factor as needed to improve efficiency.

Task 1: Map out the motors (15%) AS3_1.java

In this task you will map out the motor ranges and neutral values for your Sota robot. These *should* be fairly consistent between robots, but depending on how motors were mounted they may be a little different. I have very different results for my 2 Sotas. Further, the specs do not provide the expected ranges and values for the motor controller.

Write a small program that connects to the Sota, turns off the motors, and enters a while loop that does not quit until the user presses the power button. Inside the loop, read the robot motor positions, and report the positions to your ServoRangeTool (see below) which keeps track of the minimum and maximum positions for all the motors. Also calculate the mid-point pose, which for this robot conveniently ends up being the 0 radian, neutral pose for all joints. Save the observed mid and max ranges to a file for later loading.

Since the motors are off, you (gently) take the Sota and manually move all the motors around to their full extremities. **BE GENTLE,** as those servos are highly geared and your quick movements will produce a lot of force internally**.**

To keep things sane and avoid busylooping, use CRobotUtil.wait to slow down the loop (I do 10Hz).

Make a new object file `ServoRangeTool` which…

- Implements `Serializable`, to simplify saving and loading. Find an online tutorial, this is seriously simple. Give it a public method `Save(…)` which saves the current instance to a file,

and a `public static ServoRangeTool Load(…)` that loads the servo ranges from a data file saved using the Save method.

- Has a `public void register(Short[]pose)` method that takes a pose and updates the internal min/max/mid etc. values for each joint.
- Has three methods, `CRobotPose getMinPose(), CRobotPose getMinPose(), CRobotPose getMidPose()` that creates and returns objects representing the min, max, and mid points for all joints.
- Has a print function that does a pretty print of the ranges, e.g., mine looks like this:

```
-------------

       Body Y:         -33        -33        -33
   L Shoulder:        -319       -318       -318
      L Elbow:        -117       -116       -115
   R Shoulder:         284        284        285
      R Elbow:         135        135        136
       Head Y:           3          3          3
       Head P:         -66        -65        -65
       Head R:          20         20         21
```

Showing the min, mid, and max values. This is at program start before exploring the range which is why min almost equals max.

- For now, no other public methods are necessary apart from the constructor.

Hints:

- The hassle here is in learning to work with the `CRobotPose` and related data structures. Note that each motor has an ID that is needed in some contexts, and in others, you need the *index* of the motor in a data array. To simplify this mess, make a `TreeMap<Byte, Byte> _IDtoIndex` that you populate in your constructor to map motor IDs to indices.
  - In general you will want to pull out the `Short` array from the pose and work with that for simplicity. The above map makes that easy to work with.
- Note that `CSotaMotion` has useful constants that give the IDs for each of the robot's joints.
- I provided a `ServoRangeTool.java` that helps with some of the make-work in this part of the assignment and gives hints on structure.
- My entire AS3_1.java is 58 lines, entire ServoRangeTool (with all functions implemented) is 176 lines.

Deliverables

- **Demonstration /5 (90%)**
- **Writeup: If you repeat the operation, do you get the same values exactly? Why or why not? 1 Paragraph (10%)**

Task 2: Test the motor ranges (15%) AS3_2.java

Make a simple program, similar to Task 1, that connects to the Sota. Load the servo ranges from your file created in Task 1 (hint: `ServoRangeTool` makes this trivial), print the motor ranges, enable the motors, and make the robot move to the center position of all the motors. This should be the neutral position.

Warning: don't try to move to the min or max positions because, if all motors do this at the same time, they collide, and you can damage the robot

After the robot reaches the mid point, go through each joint in turn (following the order in `getServoAngles` is fine). For each, move the joint to the min, then the max, then back to the mid, and then move on to the next joint.

My AS3_2.java file is 81 lines.

Deliverables

- **Demonstration /5 (70%)**
- **Writeup: Did you notice a dead range in the elbows? Based on your knowledge of servo motors, why may this happen? 1 Paragraph (20%)**

Task 3: Convert motor values to radians (15%) AS3_3.java

Servo motor values are not useful outside of considering a specific motor, so we need to translate to radians to enable us to work more generally with their angles.

First, you will need to update your `ServoRangeTool` class to implement the `calcAngles` and `posToRad` methods. The result is a `RealVector` of radians where each element corresponds to the provided motor positions in order. For example, the first element will be the body, then the left shoulder, etc. Note that the `MatrixHelp` library has a print function.

To do this, you need to study the `sota.urdf` file provided in class to find the min and max, in radians, for each joint. These correspond directly to the min and max positions you found. You just need to calculate a linear mapping from your observed min/max/current motor position to a radian value.

Hint: Check out `MotorStatesLive.java` for how to print the live data in a readable and useful way.

Hint: I found it useful to construct a hard-coded TreeMap, `_motorRanges_rad`, that mapped motor ID to a `Double[]{min, max}`, to simplify the calculation.

Hint: Although the standard for robots is to be oriented to face down the planar X axis, with Z being up, for some reason the Sota is not oriented this way. Study the urdf (e.g., the relative positions of the arms, eyes, etc.) to determine the robot's orientation in world space before starting.

Hint: Pay attention to the right-handed rule to determine which way the joints rotate for positive radians. You should be able to watch the live output as you manipulate a joint to test this. If the handedness is backward, just reverse the polarity of the min/max values (such that min would be positive) in your hashmap. For the most part, the URDF does not do any frame rotations apart from joint movement so frames are oriented with respect to the origin when motors are at their neutral. Set the robot here and think in terms of the world frame.

Your output should look something like this, where the new right column shows current radian position:

```
-------------
      Body Y:     -1323        3     1330    0.27 rad
  L Shoulder:     -1539     -304      931    0.44 rad
     L Elbow:      -931     -116      698    -1.02 rad
  R Shoulder:      -877      312     1501    0.42 rad
     R Elbow:      -676      138      953    1.30 rad
      Head Y:     -1497       25     1548    0.00 rad
```

```
Head P:      -332      -86      160    -0.12 rad
Head R:      -324       21      367     0.11 rad
```

Test Cases:

- While not needed yet, solve the inverse radian to value functions at the same time. This will save time later and you can use them to test your math.
- ~~Everything at neutral position should give 0 radians.~~ Manually rotate to be sure that radians go up or down using the right hand rule.

Deliverables

- **Demonstration /5 (75%)**
- **Writeup: If you did this for several robots with natural variation, what differences in the resulting radians would you expect to see? Why? 1 Paragraph. (20%)**

Task 4: Solve the forward kinematic chains (35%) AS3_4.java

You will implement a class called `SotaForwardK` which will solve the forward kinematics for the Sota robot. From this point on things get really complex and there is a lot of data kicking around, so I have provided some helper classes. In Task 4, you will use `Frames` to simplify referring to the different frames.

The FK class is simple. The constructor takes current angles (in radians), calculates all the transformation matrices, and stores the final useful frames in a global map. To do this, you will need to start working on the `MatrixHelp` library and implement the stubs as needed. Note that you can extract the position and orientations of each frame using functions in `MatrixHelp`. My entire `MatrixHelp` file is 201 lines.

Hint: This is a hard task but the code is not that complex. The challenge of course is getting all of your transformation matrices created properly, and, multiplied in order. The key is to incrementally build and test results, and spend the time to mentally understand what to expect and how to interpret the results.

Hint: First define where the Sota base is as a trivial case (it's at the origin). Then the Sota body (just above the base). Use many intermediate variables for each step to make it easier conceptually., I have 21. At each step, do a live print of the values and move the motor while observing the origin and direction to be sure it behaves as you would expect.

Hint: here is a snippet of my code. My entire `SotaForwardK.java` is 56 lines:

```
//======== setup Transformation matrices
Matrix _base_to_origin = = MatrixUtils.createRealIdentityMatrix(4);
Matrix _body_to_base = MatrixHelp.T(...);
...
//========== precalculate combined chains
Matrix _body_to_origin = _base_to_origin.multiply(_body_to_base);
...
Matrix _l_hand_to_origin = _l_elbow_to_origin.multiply(...);
```

Hint: A big gotcha is going from the shoulder to the elbow. Although this segment is at an angle, the URDF specifies this transform without any rotation. Therefore, "forward" at the elbow will still be axis aligned and not pointing where the arm points. ~~To fix this, use a different forward direction for the arms based on your knowledge of the static direction of that arm as in the URDF.~~

Finally, work through the following test cases to convince yourself that it is working:

- Point the robot all the way to the right, and the neck all the way to the right. The robot should be able to look behind it as shown in the look dir. Do the same for the left.
- Print the hand locations. When the hands are close to the ground, the Z should be small. Measure it, your measured Z from the ground to the hand tip should be very close to what is printed.
- Calculate the distance between the two hands. As you move them around, check with a ruler the distances. They should be easily within 5mm, probably within 1mm.

I provided an example of what my live output is like:

```
-------------
      Body Y:     -1330        2     1334    0.00 radd
  L Shoulder:     -1540     -316      907    0.21 rad
     L Elbow:      -949     -126      697    -0.01 rad
  R Shoulder:      -910      293     1496    0.44 rad
     R Elbow:      -676      138      953    0.25 rad
      Head Y:     -1540       -5     1529    -0.00 rad
      Head P:      -328      -80      167    -0.03 rad
      Head R:      -365       13      391    0.09 rad
----- head  t: 0.000 0.000 0.195 1.000   ypr: -0.002 0.090 -0.033
----- lhand  t: 0.091 -0.089 0.128 1.000   ypr: -0.006 -0.001 0.208
----- rhand  t: -0.081 -0.090 0.116 1.000   ypr: 0.196 0.015 0.280
Distance between hands: 17.303651411772837
```

I provided AS3_4.java as a front-end to the kinematics libray so save some work.

Deliverables

- **Demonstration /5 (75%)**
- **Writeup: Why is the arm elbow such an exception? Explain the problem. What does the current solution do to the final frame rotation? How would you go about solving it without using Rodriques' formula? Half page+ (15%)**

Task 5: Solve the inverse kinematics (15%) AS3_5.java

This is probably the hardest task in the course, math-wise. You will implement a numerical solution to solving the Jacobian, apply a pseudoinverse to invert it, and then apply simple optimization to iteratively solve the IK problem. The key here will be to think through the stages, implement in a modular fashion, and to plan for expected output. Debug at each step.

This is really important. All of this enables Sota to do a Wax on, Wax off, motion, using the provided AS3_5.java.

The IK class is used as follows. Creating an IK instance will calculate the Jacobians *at that configuration*. You can use this for testing and debugging, but you won't use it in the end since it is used by the solver as outlined below.

To implement `makeJacobian`, first make sure you clearly understand the structure of the Jacobian and what each entry means intuitively! Simultaneously implement the O an R Jacobians since they rely on the same math. We will construct the Jacobian numerically, where we simulate the derivative by calculating a change based on a very small offset using the standard derivative formula.

Loop through each motor value that serves as input to the Jacobian

- Perturb that motor only by a very small number (e.g., 1e-10 or so) NUMERICAL_DETAL_rad (given)

- Calculate the new FK with that change, and calculate the difference in the resulting O and R vectors. Store the result in the appropriate Jacobian, and don't forget the division to normalize it.
  - $J_{ij} \approx \frac{FK(x+\Delta)-FK(x)}{\Delta}$
- **Hint:** You can work on the O and R as vector (not element-wise) and then set the matrix columns as a whole to simplify your code.
- Restore the perturbed motor value before moving on to the next. Each update should only have one value perturbed!

Use the provided pseudoinverse function to calculate the inverse of your resulting Jacobians.

Hint: Being clever with your loops, indices, and appropriate use of the Frame motor indices greatly simplifies this problem. You should be able to make a general solution for any `frameType` provided.

Once the above works, you will abandon using it directly and start to work on the optimization solver. Now, you call the static `solve` function, which implements an optimization algorithm based on the Newton-Raphson method. This doesn't need an instance since it creates many IK instances as it goes.

- While the L2 norm of the error is larger than some threshold, or, you have tried < SOME_MAX[1] times
  - Use FK to calculate the pose at the current solution. Start from the robot's live position before updating.
  - Instantiate an IK object to solve the Jacobian at the current solution. Note that you need to re-do the Jacobians at each loop iteration!
  - Calculate error from the current pose and the target pose, as a vector. This provides a per-dimension error.
  - Use the Jacobian inverse to translate the current error vector into suggested deltas to the motor positions to correct that error. This *should* move you closer to a solution. Probably.
  - Update the solution using these deltas.

Hint: Note that the standard Newton-Raphson method may diverge instead of converge, especially when far from a solution, so you should save the *lowest* error value (based on the L2 norm) and not the final one.

Hint: Interpreting the inverse Jacobians is difficult, so for debugging print the regular Jacobian to help interpret what is happening.

My `SotaInverseK.java` file is 116 lines.

Deliverables

- **Demonstration in all difficulty modes /5 (80%)**
- **Writeup: What happens if we use a large delta perterbution (say, 1), instead of a tiny one? This would save the division. Why don't we do this? Half page+ (20%)**

Code (5%)

Deliverables

Code: / 5

[1] 10-20 is a good place to start.