U of R Robotics Workshop: Inverse Kinematics

Samuel Triest

March 2020

Contents

1 Introduction

This workshop was designed as an introductory workshop to explain the concepts of forward and inverse kinematics for control of robot arms. Moving forward, we will assume knowledge of the following:

- Basics of calculus (we will be taking partial derivatives)
- Basics of linear algebra (there will be a lot of matrix math)
- Familiarity with Python 3 and numpy (this workshop was written in Python)

Those who have completed the workshop will have gained introductory knowledge of the following:

- How to parameterize and compose spatial rotations/translations as homogeneous transform matrices.
- How to parameterize robot manipulators as spatial rotations and compute the manipulator's spatial position as a function of its controls (joint angles).
- How to use basic Jacobian-based approaches to generate a set of controls that allow manipulators to reach arbitrary spatial positions.

2 Setting up the Workshop Code

2.1 Introduction

Workshop code can be found here. TODO: write about how to set everything up.

2.2 Setting up your environment

The inverse kinematics library requires numpy for the vast majority of the calculations and matplotlib for visualization. Personally, I think that the simplest way of doing all of this is via a package manager such as Conda.

2.3 An Overview of the Abstractions in the Workshop Code

The workshop code is divided into two folders, `inverse_kinematics_workshop` and `inverse_kinematics_workshop_incomplete`. The actual code-writing for the workshop will be done in `inverse_kinematics_workshop_incomplete`. `inverse_kinematics_workshop` is provided to check your solutions.

Provided in the workshop code is a number of useful classes to make the process of spinning up kinematics examples much simpler. Documentation is provided in Appendix A.

3 Forward Kinematics

3.1 The Problem

Forward kinematics is the technique used to answer the following question:

Given a set of joint angles for my arm, where is the endpoint of my arm in space?

In general, we are able to solve this problem analytically using spatial rotations and homogeneous transformation matrices.

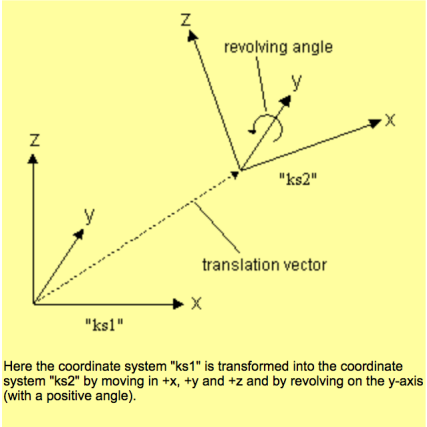

Figure 1: An example coordinate translation (from [2])

3.2 Spatial Rotations

Any point in 3D space can be represented as a 3-dimensional coordinate (x, y, z). Furthermore, points in 3D space can have orientations associated with them. Any orientation can be represented as a triple of rotations around the axes of the original coordinate frame (roll, pitch and yaw). For convenience, we will instead take orientation to be the basis vectors of the new coordinate frame. Thus, we can express a position and orientation as:

1. A three scalars d_x, d_y, d_z denoting displacement from the origin along x, y, z.
2. Three basis vectors $x,' y', z'$ that form a new, rotated coordinate system with origin at (d_x, d_y, d_z).

A spatial rotation, or a change in position and orientation, is a translation followed by a rotation. It can be used to provide a correspondence between two coordinate frames. If we consider a robot arm, in which joints can be represented as origins of coordinate frames, we can use spatial rotations to encode how the joints move with respect to other joints.

3.3 Homogeneous Transformation Matrices

Homogeneous transform matrices (HTMs) are a means of encoding spatial rotations in matrix form. A HTM is a 4x4 matrix that takes the following form:

$$T = \begin{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ x' & y' & z' \\ \vdots & \vdots & \vdots \end{bmatrix} & \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix}$$

where x', y', z' are the rotation, and d_x, d_y, d_z is the translation.
A key property of HTMs are their **composability**. That is, given two HTMs H_0^1, H_1^2 that represent transformations from frame 0 to 1 and 1 to 2, respectively, the HTM that encodes the transformation from frame 0 to 2 can be computed as: $H_0^2 = H_0^1 H_1^2$.

An example:
Let

$$T_0^1 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_1^2 = \begin{bmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus,

$$T_0^2 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.4 Robot Links as Spatial Rotations

We can express the joints in a robot arm as HTMs. Given the composability property of HTMs, this allows us to compute where our robot is in space, given the HTMs for each joint (I will be using joint and link interchangeably throughout this workshop).

We can compute the transformation matrix of a link using DH parameters.

3.4.1 DH Parameters [1]

DH parameters are a means of expressing common robot joint types more succinctly than regular HTMs (with some constraints on the types of rotations it can encode [1]). A rotation can be described by 4 DH parameters a, d, α, θ. To get the spatial rotation, we apply these steps in sequence:

1. Translate d along z
2. Rotate θ around z
3. Translate r along x
4. Rotate α around x

Here is a useful video.

There are two main types of joints that we will consider; **prismatic** and **revolute**. Prismatic joints translate linearly (like a piston), while revolute joints revolve around a single axis. By convention, a revolute joint has θ as a free parameter (i.e. a control signal can set θ to some value) and a prismatic joint has d as a free parameter.

The conversion from DH to HTM is as follows:

$$\begin{bmatrix} cos(\theta) & -sin(\theta)cos(\alpha) & sin(\theta)sin(\alpha) & a * cos(\theta) \\ sin(\theta) & cos(\theta)cos(\alpha) & -cos(\theta)sin(\alpha) & a * sin(\theta) \\ 0 & sin(\alpha) & cos(\alpha) & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

3.5 Using DH Parameters and Spatial Rotations for Forward Kinematics

Let's revisit the original question:

Given a set of joint angles for my arm, where is the endpoint of my arm in space?

We can solve this problem my taking the following steps:

1. Compute the DH parameters for your arm (leaving free variables for revolute and prismatic joints).
2. Assign joint angles to the free variables in the DH parameters.
3. Compute the rotation matrices for each set of DH parameters.
4. Compose the rotation matrices together to get the location of the arm's end effector.

3.6 Problems

3.6.1 Computing HTMS of a Robot Arm

Find `compute_htms(self)` in `kinematicchain.py`. Implement the function body by doing forward kinematics for each joint in the arm by composing the HTMS for each link in the kinematic chain. The output of this function should be a [(N+1)x4x4] numpy array, where the i-th entry in the matrix is the HTM from the base frame to the end of the i-th link. (The first entry should be the base frame = **I**)

3.6.2 Computing the End-effector Position

Find `end_effector_position(self)` in `kinematicchain.py`. Implement the function such that it returns the current position (x, y, z) of the chain's end-effector as a 3x1 numpy array. Hint: This should be easy if you did the last question.

4 Inverse Kinematics

4.1 The Problem

Inverse kinematics is the technique used to answer the following question:

Given a desired point in space, what joint angles do I need to move my arm there?

This problem tends to be more difficult than forward kinematics for a number of reasons:

- There may be zero or many solutions (If the target is too far from the arm, we can't solve. Conversely, if there are many links in an arm, there may be multiple ways to reach the same point.)
- Any closed-form solution depends on the configuration of the arm.

Given these challenges, a popular approach to inverse kinematics relies on Jacobian-based methods. This is the approach we will be using for this workshop.

4.2 Why the Jacobian is Useful

The Jacobian J of a system is a collection of the partial derivatives of system output w.r.t. the system inputs. Mathematically speaking, let's say our system has output **y** and inputs **x** (both are vectors). The Jacobian is:

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_N}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_M} \end{bmatrix} \tag{2}$$

This becomes useful for the following:

$$\frac{\partial y}{\partial x} = J \tag{3}$$

$$\partial y = J \partial x \tag{4}$$

$$\partial x = J^{-1} \partial y \tag{5}$$

What this means is that if we have some desired change in our output **y**, we can find the corresponding change in our input **x** through the inverse of the system's Jacobian.

4.3 Calculating the Jacobian of an Arm

While there are analytical solutions for computing the Jacobian of a robot arm, we can also do it numerically. Note that we can approximate the gradient of $\nabla_x f(x)$ as $(\frac{f(x+\epsilon)}{\epsilon})$ for some small value of ϵ. Since the columns of the Jacobian are the gradients of the output w.r.t. a given input variable, we just need to compute the gradients of the arm w.r.t. each joint angle. This lends itself to the following algorithm for computing the Jacobian of an arm.

Input: Array of joint angles θ, forward kinematics function $FK(\theta)$ of the arm.
Output: Jacobian J of the arm
$e_{base} = FK(\theta)$
for *Each joint angle* θ_k **do**
  $\theta[k] = \theta[k] + d\theta$
  $e_{new} = FK(\theta)$
  $\nabla_{\theta_k} = (e_{new} - e_{base})/d\theta$
  $\theta[k] = \theta[k] - d\theta$
end
$J = [[\nabla_{\theta_1}] \dots [\nabla_{\theta_N}]]$

Algorithm 1: Numerically Computing the Jacobian of a Robot Arm

One major point is since the Jacobian is a first-order approximation of the system behavior, it is only valid in a small neighborhood around the current system input. As such, we must re-compute the Jacobian every time we change our inputs. Often, we will scale our x step by a small constant η such that $\partial x = \eta J^{-1} \partial y$ for stability.

4.4 Computing Inverse of a Jacobian

A matrix is only invertable if it is square. Jacobians will only be square if the dimension of the output and input match. This is rarely true in practice. As such, we often replace the inverse of a matrix with its **pesudo-inverse**. This is computed as:

$$J^{-1} = (J^T J)^{-1} J^T \tag{6}$$

4.5 Putting it All Together

We now know how to take the inverse Jacobian of a robot arm. However, we still need to compute our desired change in system output. Fortunately, our system outputs a point in 3D space. We're given a target point (x_t, y_t, z_t). We can use forward kinematics to compute end effector position (x_e, y_e, z_e), we can set

$$\partial e = \begin{bmatrix} x_t - x_e \\ y_t - y_e \\ z_t - z_e \end{bmatrix} \tag{7}$$

∂e can also be thought of as the error vector, as it is the vector that goes from the end effector of the arm to the target point.

As such, we can perform inverse kinematics by repeatedly applying the following steps:

1. Use forward kinematics to compute the current position e.
2. Numerically compute the system's Jacobian J.
3. Take the pseudo-inverse J^{-1} of the Jacobian.
4. Compute the error vector $t - e$
5. Let $\theta_{new} = \theta + \eta J^{-1}(t - e)$

4.6 Problems

4.6.1 Computing the Numeric Jacobian

Find `numeric_jacobian(self)` in `inversekinematics.py`. Implement the function body so that it returns an ExI matrix containing the Jacobian of the chain.

4.6.2 Computing the Pseudo-Inverse of a Matrix

Find `pseudo_inverse(J)` in `inversekinematics.py`. Implement the function body so that it returns the pseudo-inverse of J.

4.6.3 Putting it all Together

Find `step(self)` in `inversekinematics.py`. Implement the function body so that it uses the Jacobian-based update method outlined in this paper to take one step of inverse kinematics.

4.7 Aside: Optimization and Gradient Descent

Optimization is a major area of academic research. As such, we will only discuss the basics of gradient descent, one such technique to solve optimization problems. We will consider optimization to be the following:

Given an objective function O [1], and a function $f(\theta)$, we want to find θ that minimizes $O(f(\theta))$ (Often abbreviated to $O(\theta)$). Mathematically, this is equivalent to:

$$\theta^* = \arg\min_{\theta} J(f(\theta)) \tag{8}$$

Gradient descent gives us a way of solving this argmin problem. Essentially, if we know the gradient of the objective function at $f(\theta)$, we can step θ in a direction that reduces J. The gradient is given as the following:

$$\frac{\partial O(\theta)}{\partial \theta} = \frac{\partial O}{\partial f}\frac{\partial f}{\partial \theta} \tag{9}$$

Note that gradient descent requires two things:

1. The derivative of the objective w.r.t to the function f.
2. The derivative of the function f w.r.t. its input θ.

4.8 Inverse Kinematics as Optimization

Roughly speaking, we can think of inverse kinematics as an optimization problem, where we use a vector of our joint angles θ as our optimization variable.

4.8.1 Objective

Our goal is to move our end effector to a target point, whose coordinates we will denote as (x_t, y_t, z_t). If we have our end effector as (x_e, y_e, z_e), we can use Euclidean distance as an objective function. That is:

$$O(x_e, y_e, z_e) = \sqrt{(x_e - x_t)^2 + (y_e - y_t)^2 + (z_e - z_t)^2} \tag{10}$$

where x_e, y_e, z_e are functions of θ.

4.8.2 Incorporating Forward Kinematics

We now have a differentiable objective. All we need now is a differentiable function that relates joint angles to end-effector position. Thankfully, this is exactly what forward kinematics does! Let's denote our forward kinematics as a function $FK(\theta)$. We can thus re-write our optimization objective as $O(FK(\theta))$.

[1] Typically, J is used to denote the objective function, but we need J to represent to Jacobian.

4.8.3 Some Math

An important thing to note is that $FK(\theta)$ is a vector-valued function. Essentially, this just means that we replace the partial derivatives with gradients.

Let's start by writing an equivalent form of our objective. Since constants get subsumed into the learning rate, we can say:

$$O(x_e, y_e, z_e) = 0.5(x_e - x_t)^2 + 0.5(y_e - y_t)^2 + 0.5(z_e - z_t)^2 \tag{11}$$

The gradient of this function is:

$$\begin{bmatrix} x_e - x_t \\ y_e - y_t \\ z_e - z_t \end{bmatrix} \tag{12}$$

5 Testing Your Solution

Provided in this workshop is a file called `testkinematics.py`. This will create an arm and a target point. It will then run your inverse kinematics code. If the end-effector is within a certain distance of the target, it will randomly pick a new target. It will do this for a fixed number of timesteps, and make a video of the arm. You can run this with `python testkinematics.py`

References

[1] "Forward kinematics: The denavit-hartenberg convention." [Online]. Available: https://users.cs.duke.edu/~brd/Teaching/Bio/asmb/current/Papers/chap3-forward-kinematics.pdf

[2] "Basics of coordinate metrology unit 3: Coordinate systems in space - transformation." [Online]. Available: https://www.aukom.info/fileadmin/Webdata/el/english/elearning/03/ks3_transformation.htm

A Documentation

- `Link`: A single link of a robot manipulator. For the purposes of this workshop, we'll assume that all of our links are **revolute**. That is, the control signals for all of our joints will be angles θ. (Think like your elbow or knee). This differs from **prismatic** joints, which are controlled through translations (like a piston). Links have the following:
  1. `__init__(a, alpha, d)`: This initializes a `Link` with DH parameters `a, alpha, d`. We also initialize `theta` as 0. Arguments for `theta` are not included as theta varies as control signals are sent to the joints.
  2. `htm()`: This function automatically generates the homogenous transform matrix for the `link`, given its current set of DH parameters.
- `FixedLink`: This is a child class of `Link`. Unlike `Link`, `FixedLink` is not intended to have any degrees of freedom (i.e. `theta` is fixed as well). FixedLinks have the following:
  1. `__init__(a, alpha, d, theta)`: This initializes a `Link` with DH parameters `a, alpha, d, theta`.
  2. `htm()`: This function automatically generates the homogenous transform matrix for the `link`, given its current set of DH parameters.
- `KinematicChain`: An interface for performing forward kinematics on a series of links in sequence. Implementing some of the functionality of this class is part of the workshop.
  - Fields:
    * `links`: An ordered list of the links that comprise this `KinematicChain`, from base to end-effector.

    * `control_links`: An ordered list of only the links in this `KinematicChain` that are revolute joints (i.e. are of class `Link` and not `FixedLink`). Links in this class are also ordered from base to end-effector.
    * `control`: A `numpy` array containing the current control vector of the `KinematicChain` (i.e. the `theta` values of all the Links in `control_links`).
  - Methods:
    * `append_link(link)`: Adds a new link to the end of the `KinematicChain`.
    * `update_control(controls)`: Assigns a new control vector to the `KinematicChain`.
    * `compute_htms()`: Given the current kinematic chain, computes a tensor of dimension [Nx4x4], where N is the number of links in the `KinematicChain`. The i-th element of this tensor should be the homogeneous transform matrix that parameterizes the spatial rotation/translation from the origin to the current position of the i-th link in the `KinematicChain`.
    * `end_effector_position()`: Computes the x, y, z location of the end-effector of the `KinematicChain`.
    * `render()`: Draws the `KinematicChain` in 3D-space.

- `InverseKinematicsSolver`: Performs inverse kinematics on a `KinematicChain` to specify controls to arbitrary x, y, z points.
  - Fields:
    * `chain`: The `KinematicChain` to perform inverse kinematics on.
    * `control_dim`: The dimension of the control signal for `chain` (i.e. the number of `Link` in `chain`).
    * `lr`: The learning rate, i.e. how strongly we update our control vector w.r.t the gradient.
    * `dt`: Perturbation parameter used to numerically compute the local Jacobian.
    * `dt_max`: An upper-bound on join velocities for the arm.
    * `convergence`: A threshold (as Euclidean distance between the end-effector and target point) under which the problem is considered solved.
    * `target`: A point x, y, z that the `KinematicChain` is trying to move its end-effector to.
  - Methods:
    * `set_target(point)`: Sets `target` to `point`.
    * `sample_target(bounds)`: Sets `target` to a random point within `bounds`, where `bounds` is a dictionary `{'x':(a, b), 'y':(a, b), 'z':(a, b)}`, where `(a, b)` are the lower and upper bound for their respective coordinate.
    * `numeric_jacobian()`: Computes the local Jacobian of the `KinematicChain` in its current position.
    * `pseudo_inverse(matrix)`: Computes the pseudo-inverse of `matrix`.
    * `error()`: Computes the spatial distance between the end-effector position of `chain` and `target`.
    * `step(point)`: Performs one step of inverse kinematics.
    * `converged()`: Returns `True` if the end-effector is within `convergence` of `target`. Otherwise returns `False`.
    * `render(bounds)`: Renders `chain` and `target` in 3D space.
    * `make_video(render_every, resample_every, steps)`: Solves a number of inverse kinematics problems in sequence, and saves a video. `render_every` dictates the number of steps taken before a frame is saved. `resample_every` determines the maximum number of steps the solver has before `target` is resampled (`target` is automatically resampled if solved). The simulation terminates after `steps` steps.