1 Introduction: Modeling Robots

A number of modeling concerns are shared across (virtually) every robot we can build or imagine.

1.1 Relevant Spaces

A robot exists in the physical world, and we need to keep track of where it is and how it's moving. A **state space** vector x is used to capture every important element of a robot's state. For example, a robot that sits on the 2D floor in some position and has a forward direction that can rotate is often modeled as $x = [x, y, \theta]^T$[1]. The end of a robot's manipulator takes some position in 3D space and its orientation also exists in 3D, often $x = [x, y, z, roll, pitch, yaw]^T$, with common short-hand 6-DoF (but we'll talk soon about the big limitations of this choice).

The same robot need not always be modeled in the same state space. My favorite example is the rubic's cube. If we want to throw the whole cube to our friend as a solid object, then "only" 6-DoF is needed, or perhaps we can even simplify to track only the position of the centre of mass. However, to solve the cube, we need to track the positions and rotations of the 26 colored cubes – how you'd model this exactly is left as an exercise.

A robot may arrive at its state space using any number of internal **configuration space** variables, which are commonly called θ. The Gen3 lightweight arm that we will use throughout the course has six rotatable joints: $\theta = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]^T$.

1.2 Kinematics

The problem of mapping states to configurations is known generally as kinematics, and two large and related sub-problems are the key functions.

Forward kinematics takes a configuration vector as input and returns the corresponding state space vector.

$$x = fkine(\theta) \tag{1}$$

Computing fkine typically requires a number of constants about the robot such as the length of its axes, radii of wheels and more. This is a lot of book-keeping, so it's important to follow accepted community standards. Famous ones include Denavit-Hartenberg `https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters`, URDF, Unified Robot Description Format, .urdf, Mujoco's configuration format, .mjcf, and more.

Once all the critical information is loaded, the nature of most robot kinematics makes fkine a "trivial" computation, but this is not always the case. For

[1] There is an obvious terrible ambiguity about the symbol x being used as the vector's name and one of the elements. It's terrible, but so commonly used and annoying to avoid that avoiding it would cause more harm than it helps. Plug your nose and continue!

large chain robots with parallel paths, we might even need to confirm that the input θ is valid (e.g., that all physically linked chain elements really touch).

A simple example of forward kinematics is the pendulum, attached to the world at position $[0, 0]$, and measuring the angle such that it reads zero when the pole is straight up. The state space is the $[x, y]$ position of the tip of the pendulum, which can sweep out a circle of radius r. A single configuration variable, $\theta = [\theta_1]$ controls the rotation and

$$x = -sin(\theta_1) \tag{2}$$
$$y = cos(\theta_1). \tag{3}$$

A nice sequence to work through from here is:

- double pendulum (second pole attached at $[x, y]$ solved above, and can rotate by θ_2)
- cartpole (the single pendulum's base is attached to a block that can slide along the x axis, with the cart's position being a second configuration element
- cart double pendulum (or double cartpole), combining these two.

In all cases, fill in the lengths and parameters with sensible variables, l_1, l_2, h_1 as needed.

Now it's time for the fun to begin! **Inverse Kinematics** is the problem of solving for configurations that bring the robot to a desired state space vector:

$$\theta = ikine(x) \tag{4}$$

Inverse Kinematics is a search problem. Many robots have no simple closed-form solution and there can be multiple satisfactory answers (e.g. double pendulum can reach the point $[l_1, 0]$ when $l_1 = l_2$ by being "elbow-up" or "elbow-down"). Inverse Kinematics is also a crtical and inside-loop element of controlling a robot in state space. The robot may be directly commanded by a user to reach the desired states or move in the desired directions. But it's actually worse when we want to plan an optimal sequence of states and want to search over many possibilities – each branch in a search tree can require one or many calls to *ikine*.

A common solution is to implement a differential form of kinematics, such that we can apply small updates, use informed initial guesses and control the effort invested in convergence. Jacobian-Transpose methods are common, with more nice reading here: `https://www.cs.cmu.edu/~15464-s13/lectures/lecture6/iksurvey.pdf`.

1.2.1 Kinematics Exercises

(E1.1) Using pen-and-paper write down the inverse kinematics, for the series of robots: pendulum, cartpole, double pendulum and double cartpole.

(E1.2) Code ikine for the double cartpole in your favorite language. Test the solutions for a number of points and check the accuracy by running fkine on the solutions. Is your method exact? What is the run time. Good code may be very fast on a modern computer, but add good profiling or a loop counter to give you some better feedback on what's happening.

2 Dynamics

Now the fun continues! Kinematics models a robot as a static collection of parts. It neglects critical physical elements like inertia, momentum, gravity and contact. Of course these all matter for a robot, and **Forward Dynamics** is the first related problem. To model motion over time we introduce time derivatives $\dot{x} = \frac{\partial x}{\partial t}$, the system's state space velocity and $\ddot{x} = \frac{\partial^2 x}{\partial t^2}$, the state space acceleration, as well as (optionally) a control, related to force u, and compute

$$\ddot{x} = fdyn(x, \dot{x}, u). \tag{5}$$

We'll need even more physical constants to compute $fdyn$. Common fields in a dynamics configuration file include intertial matrices, centre-of-mass information, friction coefficients, joint stiffness, wind resistance, and motor constants. Electrtical properties such as resistance and capacitance are also very relevant but we'll mostly ignore these in CS to avoid zapping ourselves.

A friendly version of dynamics is provided by Newton's Three Laws. For example $F = ma$ (equivalently $u = m\ddot{x}$ in our notation and assuming u is an immediate force, which is an over-simplification) appears helpful. For a very simple system, we can indeed solve for $\ddot{x} = u/m$, but this will almost never be enough for any interesting robot. The input control is typically not a simple force, but rather a motor effort A friendly version of dynamics is provided by Newton's Three Laws. For example $F = ma$ (equivalently $u = m\ddot{x}$ in our notation and assuming u is an immediate force, which is an over-simplification) appears helpful. For a very simple system, we can indeed solve for $\ddot{x} = u/m$, but this will almost never be enough for any interesting robot. The input control is typically not a simple force, but rather a motor voltage that converts to torque in a complicated fashion. There are external forces such as gravity that do not combine in any immediate fashion. All elements vary in nonlinear ways over the state space. Perhaps most critically, for robots made of many rigid links, computing the implicit forces at joints becomes complex or unsolvable.

It is more common to apply the mechanical principles of Lagrange for practical robots. This method begins by forming a **Lagrangian**, defined as the difference between the kinetic and potential energies, a function of the state, velocity and (not in most cases, but for completeness) time: $L = L(x, \dot{x}, t) = T - U$. Notably, x here is allowed to be (abusing notation further), the Generalized Coordinates, of the system, a great potential simplification. Our interest in this expression is due to the *Lagrange equations of motion*:

$$\frac{\partial L}{\partial x_i} - \frac{d}{dt}\frac{\partial L}{\partial \dot{x}_i} = 0. \tag{6}$$

This represents the starting pathway to derive the time-varying motions for robots ranging from simple pendula to complex sets of arms. This expression (and close relative, the Hamiltonian) is valid for general classes of physical systems, including fluids, relativistic, quantum and hybrids. This is the following recipe to derive motion equations:

- Choose generalized coordinates.
- Write down expressions for kinetic and potential energy, T and U.
- Take partial and time derivatives to express the two terms in equation (6).
- Write out the Lagrange equation and simplify (solving for $\ddot{x}$ for the pure $fdyn$ form written above, or however you like for your own analysis.

The most common first example is a one dimensional spring, which opposes movement away from a set-point with force $F = -kx$, with kinetic energy based on the momentum of a simple mass, $T = \frac{1}{2}m\dot{x}^2$, and potential energy based on the stretch of the spring, $U = \frac{1}{2}kx^2$. Use the recipe above to ensure you reach $m\ddot{x} + kx = 0$, an ordinary differential equation that can be solved in time with a periodic function (e.g. $x = A * sin(ft + p)$).

The recipe above is by far the easiest part of forward dynamics simulation. To solve for a trajectory of robot motion, we need to integrate the resulting equations carefully. The general process follows classical ideas from Newton and Euler, but the software that performs general solution is being developed in a highly active way, is a potential target for deep learning(!) and consumes Billions of dollars annually. We will wade into this only tangentially, with better resources to be found in McGill's Animation courses or via internship at Pixar.

Assuming we can solve the forward dynamics generally, we get great ways to analyze and understand our robot's motion. We can sometimes plot things like a phase-space diagram showing the relationship of x with $\dot{x}$ that allows tracing of trajectories and orbits and solving dynamic planning and control by eye. This course will spend a lot of time on the algorithms to allow solution of the same in concrete software, and in the general case.

(E1.3) Solve for the equations of motion of a simple pendulum under no external force, trace the phase space vector field.

(E1.4) Solve for the equations of motion of some of the pendulum-double cartpole family. A good resource for this is `https://www.ias.informatik.tu-darmstadt.de/uploads/Publications/phd_thesis_deisenroth.pdf`

An even more challenging problem, which begins to reach "Robotic General Intelligence" in the general case is **Inverse Dynamics**. This requires solving for the control action that move the physical system as desired:

$$u = idyn(x, \dot{x}, \ddot{x}) \tag{7}$$

For very simple systems, a process known as feedback-linearization allows inversion of the feedback matrix, formed by partial differentiation. Most interesting robotic systems do not admit good solutions of this form. In general, inverse dynamics can have multiple solutions and especially when contact is involved, poses a difficult constrained optimization problem. We will mostly end up solving this with deep learning, but it's critical that we know where our problems come from, so we'll come back to the *idyn* problem itself several times.

(E1.5) Consider what conditions must be met for feedback linearization to succeed. Include analysis of the state and control dimensions as well as linear algebra properties row and column rank of the linearized control system.

3 Conclusion

This lecture has been the most surface level and basic that we'll see in the course. Its job is just to introduce (some of) the cast of symbols and problems that we'll face during the term. We'll already begin next week by thinking about sensing, uncertainty and being much more detailed with the software and computations involved.