# Robotic Arm Position Simulation - A First Principles Analysis

Nathaniel Bechard, Ivan T. Ivanov

April 30, 2023

## Contents

## 1 Abstract

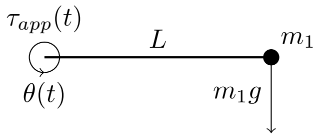
Robotic arms are one of the most common projects for makers and hobbyists. While extensive modeling and simulation is done for robotic arms in industry, hobbyists tend to shy away from simulation. This makes the transition from the hobby world to the industrial one difficult. If hobbyists have a comprehensive and simple explanation of robotic arm modeling, they will become accustomed to mathematical modeling and may come to find it enjoyable.

## 2 Introduction

This paper provides an explanation of how to model robotic arm joints; it requires readers to possess knowledge of Newtonian mechanics and ordinary differential equations. It begins with the simple 1-joint problem, which models the behavior of one mass rotated by a motor. Next, the model is generalized for a system with n joints, the robot arm.

## 3 The 1-joint problem

In this section, the equations of motion for a single motor, with a single mass, will be derived. The motor applies a torque $\tau_{app}(t)$ on a massless rod of length L. A point mass with a mass $m_1$ is connected to the end of the rod.



### 3.1 Deriving the equations of motion for the 1 joint problem

The angular acceleration of the system can be determined with Newton's second law of rotational motion: $\sum_i \tau_i = I\alpha$. We can determine the net torque and rotational inertia for our particular system:

$$\sum_i \tau_i = \tau_{app}(t) + \tau_{m_1 g}(t)$$

The torque applied by $m_1$ can be determined using the torque formula: $\tau_{m_1 g}(t) = L * F * cos(\theta)$. The net torque is therefore:

$$\sum_i \tau_i = \tau_{app}(t) - m_1 g L cos(\theta(t)) L$$

The rotational inertia term, I, can be rewritten as $m_1 * L^2$, and the angular acceleration can be turned into the second derivative of the angular position. After reorganizing terms, we get:

$$\theta''(t) = \frac{\theta_{app}(t) - m_1 g L cos(\theta(t))}{m_1 L^2}$$

Adding a friction term, B, changes the equation slightly. This term is approximate, as friction cannot be modeled with a constant and the $\theta'(t)$ term should in reality be a $(\theta'(t))^2$ term. For low velocities, a $\theta'(t)$ term is sufficient.

$$\theta''(t) + B\theta'(t) = \frac{\theta_{app}(t) - m_1 g L cos(\theta(t))}{m_1 L^2}$$

We'd like to find an analytical solution to this equation, where $\theta(t) = ...$, but, it is impossible in this case. This is not an issue, however, as numerical methods can be used. To solve numerically, is helpful to disintegrate this equation into a system of first-order equations. This is done with a simple variable substitution:

$$\theta = y_1 \ \theta' = y_2$$

$$y_1' = y_2 \qquad\qquad y_2' = \frac{\tau_{app}(t) - m_1 g L cos(y_1)}{m_1 L^2} - By_2$$

We will use this system of two first-order differential equations to obtain the numerical solution.

## 3.2 Programming the 1 joint problem

While the code can be accessed on GitHub here, an explanation of the important parts of the code is in this document.

To begin, simulation parameters are defined.

```
L = 1 #in meters, also known as L
B = 0.2 #rotational friction
Mass_1 = 1 # in kg, also known as M
motor_torque = 10 # in Nm, also known as Tq(t)
grav = 9.81 # m/s^2
simulationDuration = 20 # seconds

#initial conditions
initial_angle = 0 #rad
initial_angular_velocity = 0 #rad/s
initial_values = [initial_angle, initial_angular_velocity]
```

Next, the differential equation is turned into a function that takes in the state of the system: $\vec{Y}$, and outputs its derivative: $\vec{Y}'$. The input vector has components $y_1 = Y[0]$ and $y_2 = Y[1]$. This defines the behavior of the system.

```
def slopes(t, Y):
    return([Y[1],
          (motor_torque - Mass_1*grav*L*np.cos(Y[0])) / (Mass_1*L**2) -
               B*Y[1]
          ])
```

The function is passed through solve_ivp, which solves the differential equation numerically. In this case, it uses a Runge Kutta 7-8 solver. The initial conditions are also plugged in, in vector form.

```
sol = solve_ivp(slopes, [0, simulationDuration], initial_values, t_eval
    =T, method = 'DOP853', rtol=1e-8, atol=1e-8)

for x in range(len(sol.y[0])):
    print("position: " + str(sol.y[0, x]) + "velocity: " + str(sol.y[1,
        x]))
```

Finally, the solution is animated. The angular position of the system over time is converted to an easy-to-plot vector of Cartesian coordinates. These are fed through an animation of the 1-joint problem.

```
x1 = np.cos(sol.y[0]) * L #convert angular solution coordinates to
    cartesian for animating
y1 = np.sin(sol.y[0]) * L

print(len(x1)) # prints how many frames the simulation has

def animate(i):
    x = [0, x1[i]]
    y = [0, y1[i]]
    line.set_data(x, y) # make a line between (0,0) and (x1[i], y1[i])
    dots.set_data(x, y) # make dots at (0,0) and (x1[i], y1[i])
    txt.set_text('Time = ' +
    '{:4.1f}'.format(i*(simulationDuration/len(x1))) +
    's' + "  Velocity: " + '{:4.1f}'.format(sol.y[1,i])
    + " rad/s") # display time and velocity
    return line, dots, txt

# call the animator.
anim = animation.FuncAnimation(fig, animate, init_func=init,
frames=len(x1), interval=10, blit=True)
# each frame iterates i by one. There is a 10ms delay between frames.
plt.show()
```

## 3.3 Some Examples...

Before moving on to multi-joint systems, let's investigate the properties of the one-joint system. The differential equation for this system provides valuable insight not visible with a formulaic understanding of mechanics.

**No torque** When the motor has no torque, the system behaves as a simple pendulum. It creates the characteristic whirlpool vector field.

| System property | |
|---|---|
| Length (L) | 1 m |
| Mass1 ($m_1$) | 1 kg |
| Torque | 0 Nm |
| Friction | 0.2 |

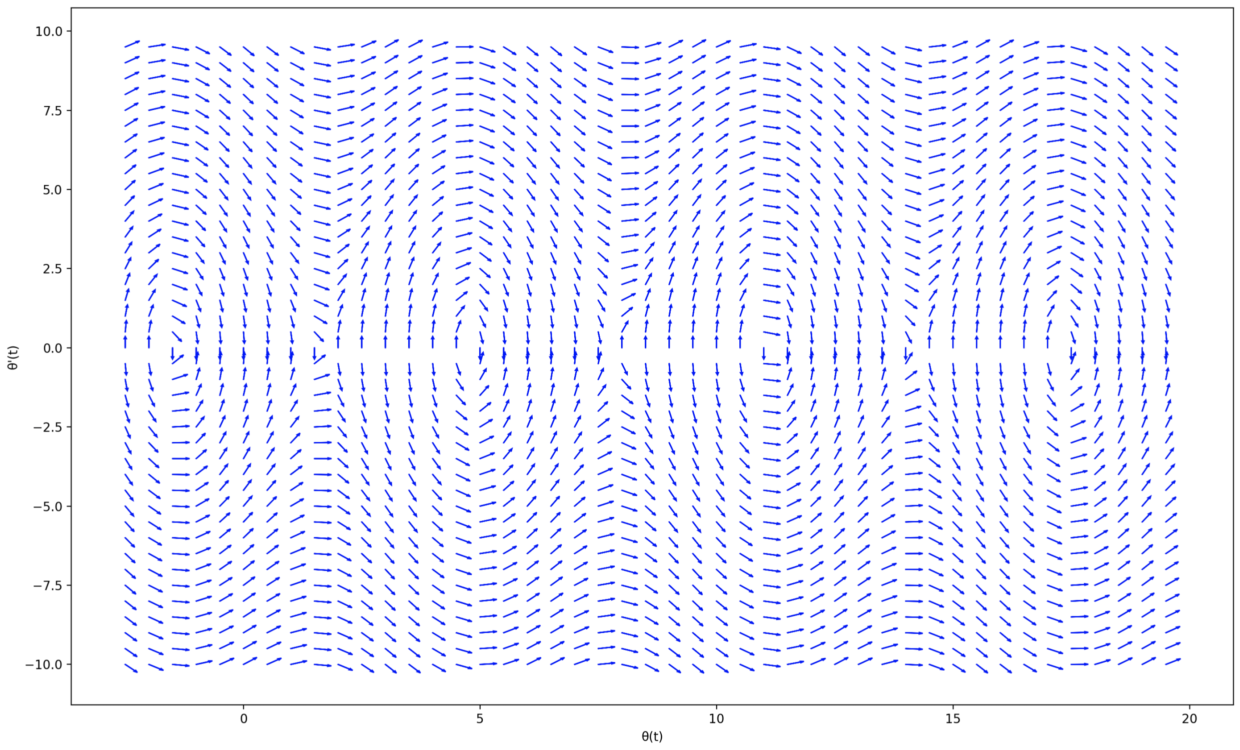


Figure 1: Vector field of a pendulum

**Damped-driven 1-joint problem** Here, the mass is acted on by a torque of 50Nm. This is just enough to make it overcome the force of gravity: $9.81N * 5m = 49.05Nm$ when the bar is horizontal. The mass accelerates until it reaches $4 \pm 0.5 \frac{rad}{s}$. This is the terminal velocity of the system, where the energy input by the motor is consumed entirely by friction. The steady-state velocity will vary between 4 and 5 radians per second, depending on whether it is fighting gravity or working with it.

| System property | |
|---|---|
| Length (L) | 5 m |
| Mass1 ($m_1$) | 1 kg |
| Torque | 50 Nm |
| Friction | 0.5 |

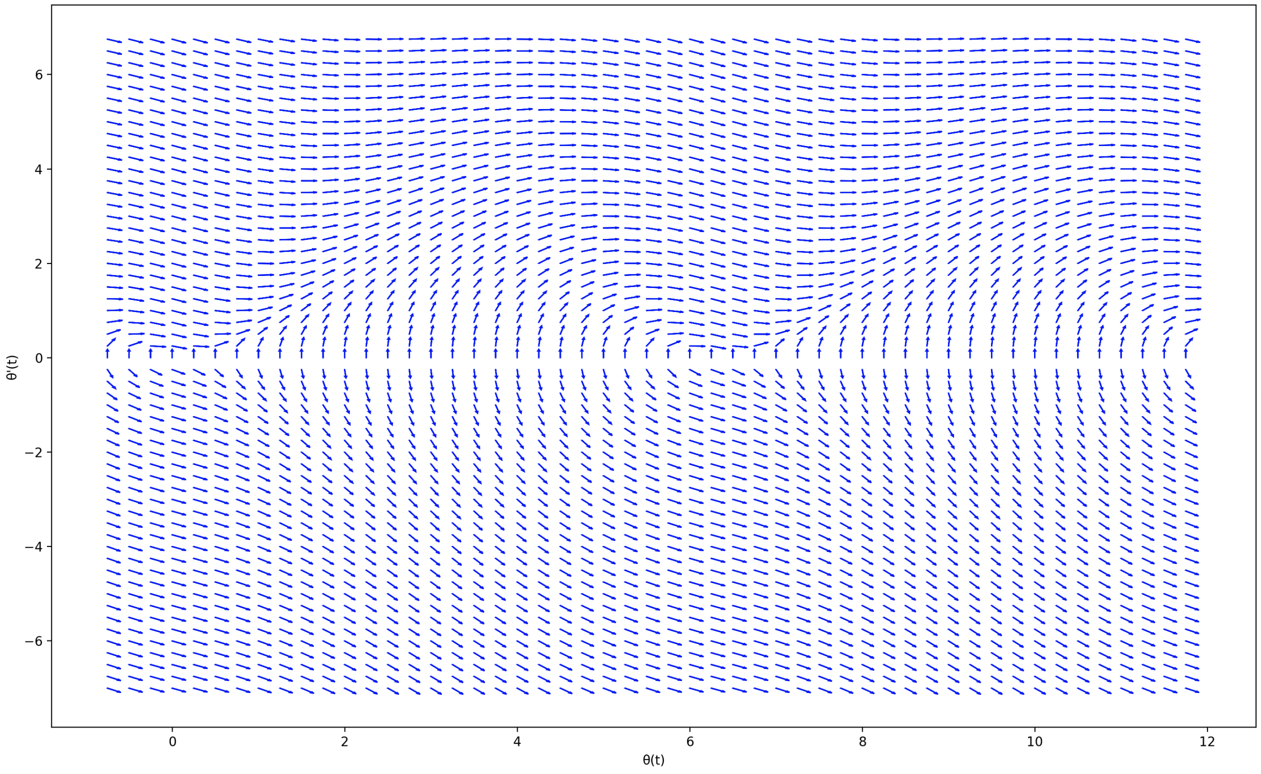


Figure 2: Vector field of a driven 1-joint system

**Under-Powered Motor** When the motor doesn't have enough power to counter the force of gravity, it will not start spinning the mass. However, the wells of the vector field can be escaped with enough potential or kinetic energy. This will result in the mass rotating forever. Notice that the centers of the wells are at $-\pi/2 + 2\pi n$, with an angular velocity of zero. This is a state with no potential energy, where the arm is pointing downwards. When velocity is high enough above the well, or the position is far enough from downwards, the solution diverges.

| System property | |
|---|---|
| Length (L) | 1 m |
| Mass1 ($m_1$) | 1 kg |
| Torque | 7 Nm |
| Friction | 0.2 |

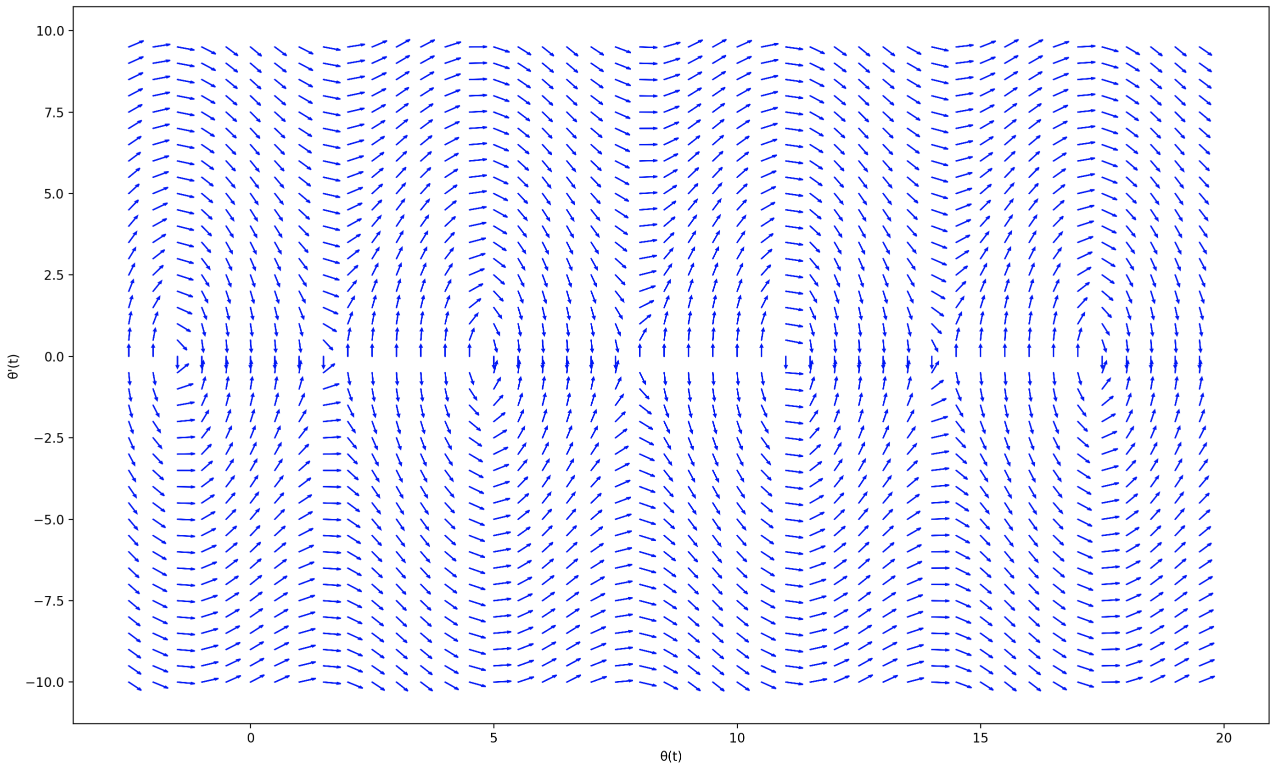
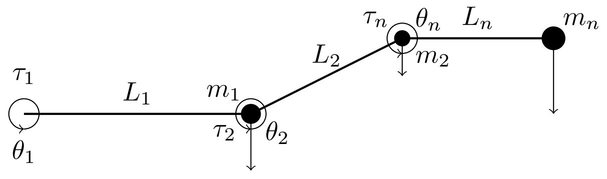


Figure 3: Vector field of a pendulum

# 4 The robot arm

This section covers the derivation and simulation of the equations of motion for an arm with an arbitrary number of joints. The robot arm is modeled as a collection of 1-joint problems organized sequentially:



Note that $\tau_1 = \tau_1(t)$, time dependencies have been removed for simplicity.

## 4.1 Deriving the equations of motion for an arbitrary number of sequential 1-joint problems

**Rotational Inertia** Finding the rotational inertia and the torque exerted by gravity becomes complicated when more joints are involved. These quantities need to be computed with respect to each joint. For joint a in the system:

$$I_a = m_n L_{a,mn}^2 + m_{n-1} L_{a,mn-1}^2 ... + m_a L_a^2 \ ,$$

where $I_a$ is the inertia calculated with respect to joint A, and $L_{a,mn}$, is the distance from joint a to the last mass. $L_{a,mn-1}$ is the distance from joint a to the second last mass. Only masses past the joint count toward its rotational inertia. The lengths used in the inertia formula are computed by treating the 1 joint problems downstream of the joint as vectors of magnitude $L_i$ and direction $\theta_i$. These vectors can be summed to calculate the net vector. The vector below is computed from joint a (which we're calculating the inertia WRT to) to joint b, a joint we're trying to find the distance to.

$$\vec{v}_{a,b} = J_a + J_{a+1} + ... + J_b,$$

$$\text{where } J_a = \begin{bmatrix} L_a cos(\theta_a) \\ L_a sin(\theta_a) \end{bmatrix} \qquad |\vec{v}_{a,b}| = L_{a,b} \qquad \hat{v}_{a,b} = \theta_{a,b}$$

The rotational inertia with respect to a joint in the system can be computed from this.

$$I_a = m_n|\vec{v}_{a,n}|^2 + m_{n-1}|\vec{v}_{a,n-1}|^2 + ... + m_A L_A^2$$

**torque applied by gravity** With the quantities just developed, the torque applied by gravity WRT to joint a can be computed:

$$\tau_a = m_a{*}L_a{*}cos(\theta_a){*}g{+}m_{a+1}{*}L_{a,a+1}{*}cos(\theta_{a,a+1}){*}g{+}...{+}m_b{*}L_{a,b}{*}cos(\theta_{a1,b}){*}g$$

**Equations of motion** The equations of motion of this system are a $2n$th order differential equation. Since this equation is nonlinear, its matrix form is quite messy. It will instead be written as a system of first-order equations:

$$\theta_1 = y_1 \quad \theta_1' = y_2 \quad \theta_2 = y_3 \quad \theta_2' = y_4 \quad .... \quad \theta_n = y_{2n-1} \quad \theta_n' = y_{2n}$$

$m_1 gLcos(y_1)$ and $m_1 L^2$ used in the 1-joint problem are now $\tau_1$ and $I_1$, which are functions of $\theta_1, \theta_2, ..., \theta_n$

$$y_1' = y_2 \qquad y_2' = \tfrac{\tau_{app1}(t)-\tau_1}{I_1} - By_2$$

$$y_3' = y_4 \qquad y_4' = \tfrac{\tau_{app2}(t)-\tau_2}{I_2} - By_4$$

$$...$$

$$y_{2n-1}' = y_{2n} \qquad y_{2n}' = \tfrac{\tau_{appn}(t)-\tau_n}{I_n} - By_{2n}$$

## 4.2 Programming and simulating the robot arm

A function that calculates the rotational inertia and torque applied by gravity with respect to each joint is defined:

```
def slopes(t, Input):     # Y is a vector of shape [x1, x2,..., xn]
    output = np.zeros(sn*2)

    #computes the distance between joints in an arm
    for joint_computed in range(sn):  #iterate joints
        rotational_inertia = 0 #rotational inertia WRT curr. joint
        net_torque_on_joint = 0 #net torque WRT curr joint
        for mass_computed in range((sn-1),(joint_computed-1),-1):
            lx, ly = 0, 0
            dist = 0
            #the vector sum
            for y in range(joint_computed, (mass_computed + 1)):
                lx = lx + Lengths[y]*np.cos(Input[y*2])
                ly = ly + Lengths[y]*np.sin(Input[y*2])
            dist = np.sqrt(lx**2 + ly**2) #The distance from Joint A to
                 B
            #The rotational inertia of the point mass computed is added
                 to the system:
            rotational_inertia += Masses[mass_computed] * dist**2
            net_torque_on_joint += Masses[mass_computed]*grav*dist*(lx/
                dist)
            # lx/dist = adj/hyp = cos
        #take the inertia and torques calculated, and use them to
            compute the ODE:
        output[joint_computed*2] = Input[2*joint_computed + 1]
        output[joint_computed*2 +1] = ((Torques[joint_computed] -
            net_torque_on_joint) / rotational_inertia)  - Bs[
            joint_computed]*Input[2*joint_computed + 1] #x2'
    return(output)
```

Note: sn is the number of segments in the arm

The simulation parameters are now vectorized, and they can be of an arbitrary length:

```
sn = 4

Lengths = [2, 1, 0.75, 2] # lenghths (m)
Bs = [0.5, 0.5, 0.5, 0.25] # friction terms
Masses = [3, 5, 1, 3] # mass terms (kg)
Torques = [300, 3, 50, 5] # torque terms (Nm)
Init_values = [0.1,0.3,0.1,0.3,0.1,0.3,0,0] # position1, velocity1,
    position2, velocity2, ....., positionn, velocityn

grav = 9.81
simulationDuration = 20 # seconds
```

The animation code is very similar to the 1-joint problem. The full code can be found here

### 4.3 Simulations

The n-joint arm exhibits interesting behavior. It can be played with extensively using the source code here

- Inertia demo - double and triple pendulums

## 5 Experiment Ideas

- Torque as a function - let the joints act as an arm - Nick's DC motor integrated in the ODE

## 6 Conclusion

The dynamics of a robot arm can be deconstructed into simple one-joint problems, which are 2nd order differential equations derived from basic laws of physics. The simulation of an arm is a demonstration of how a model based on basic laws, such as $F = ma$ and $\tau_g = mgrsin(\theta)$ can generate intricate emergent behavior. To a reader entering the field of engineering, this paper demonstrates a transition from rudimentary mathematical formulas, with no clear use in the real world, to real-world models that deepen our understanding of the universe. This will hopefully reveal the beauty of mathematics to hobbyists and makers.

## 7 References

To discuss with teacher