Adv. Software Trainings
3/11/21 Recap Document

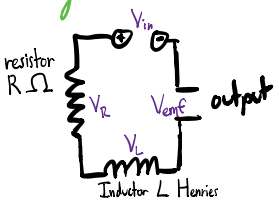
Topics
- Motor Modeling
- GoogleTest Arm Simulator → see associated source files

Motor Modeling

It is important to know the mathematical behavior of the systems which we are trying to control or model.

Most of Valkyrie's systems are powered by a motor, so lets start there.

Motors can be quite simple electrically: (if you don't get the electronics just trust the math)



We think of DC motors as a resistor, inductor, output, and voltage source. The is some voltage V_L across the inductor, V_R across the resistor, and V_{emf} across the output.

We also take as a given the following:

$V_{emf} = \frac{1}{K_v}\omega$ (V_{emf}: emf voltage; K_v: velocity constant; ω: angular velocity)

$\tau = K_t i$ (τ: torque; K_t: torque constant; i: current)

And, of course, $V_R = iR$, $V_L = L\frac{di}{dt}$.

Voltage Side
- Voltage must drop to 0 across a loop so

$V_{in} - V_{emf} - V_R - V_L = 0$ (henceforth V)

$V - \frac{1}{K_v}\omega - iR - L\frac{di}{dt} = 0$ (inductance effects are ignored)

henceforth $\dot{\theta}$

$\frac{1}{R}V - \frac{1}{RK_v}\dot{\theta} = i$

Current Side
- Current is conserved around the loop

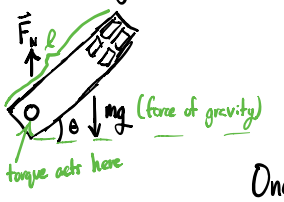
$i = \frac{\tau}{K_t}$

$\Rightarrow \frac{1}{R}V - \frac{1}{RK_v}\dot{\theta} = \frac{\tau}{K_t}$

We will be solving for part of τ (torque)

$\frac{K_t}{R}V - \frac{K_t}{RK_v}\dot{\theta} = \tau$

Now that we're here we have to consider the mechanical model of the system we are considering: an arm



The arm has forces acting on it from gravity, the normal force opposing it, and the torque from the motor. $F_{net} = 0$, obviously. τ_{net}, is more complex:

$$\tau_{net} = \tau_{motor} - \tau_g = \tau_{motor} - \frac{\ell}{2} mg\sin\theta$$

One other piece must be slotted in: the gearbox. The gearbox sits between the arm and the motor and transforms the torque and speed:

$$\tau_{out} = \frac{1}{G}\tau_{motor}$$
$$\dot{\theta}_{out} = G\dot{\theta}_{motor}$$

where G, of course, is the "gear ratio" — the ratio of gear teeth.

So, the next piece is to incorporate this into our electrical model, with one caveat: typically, we use $\dot{\theta}_{out}$ as what we're measuring, even though we measure $\dot{\theta}_{motor}$, typically we use $\dot{\theta}_{arm}$ (a.k.a $\dot{\theta}_{out}$) in our math.

$$\frac{K_t}{R}V - \frac{K_t}{RK_v}\dot{\theta}_{motor} = \tau_{motor}$$

all θ is now θ_{arm}

$$\frac{K_t}{GR}V - \frac{K_t}{G^2RK_v}\dot{\theta}_{arm} = \tau_{arm} = J\ddot{\theta}_{arm} + \underbrace{\frac{\ell}{2}mg\sin\theta}_{\text{ignore this for now}}$$

$$-\frac{\ell}{2J}mg\sin\theta + \underbrace{\frac{K_t}{JGR}}_{\text{now } C_1}V - \underbrace{\frac{K_t}{G^2JRK_v}}_{\text{now } C_2}\dot{\theta}_{arm} = \ddot{\theta}_{arm}$$

Now that we have this, we can convert it into a more useful form. Consider that what we control is V. So, we control $\ddot{\theta}$ directly (i.e without delay). So, all we need to know the position & velocity of the arm in the future is know what they are now, and the voltages we'll apply in the future. Call the pair position & velocity the "state" and voltage the "input". We want to explicitly write out that knowledge: write the change in state in terms of current state & voltage.

Review matrix multiplication as necessary.

$$\underbrace{\dot{X}}_{\text{change in state}} = \begin{bmatrix} 0 & 1 \\ 0 & -C_2 \end{bmatrix}\underbrace{\begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}}_{\text{current state}} + \begin{bmatrix} 0 \\ C_1 \end{bmatrix}V - \frac{\ell}{2J}mg\sin\theta$$

This is called the "state-space" model of the system. This one is called a "continuous" model as time is considered infinitely divisible. However, in our code, we can only calculate so often. So we must consider the "discrete" model. In this case we want to know the state one timestep (Δt) forwards. We denote the current state as $X[k]$ and the next as $X[k+1]$. Also,

$$\dot{\theta}[k+1] = \ddot{\theta}[k]\Delta t + \dot{\theta}[k]$$
$$\theta[k+1] = \dot{\theta}[k]\Delta t + \theta[k]$$
} this should be familiar as "position = velocity * time"

$$X[k+1] = \begin{bmatrix} 1 & \Delta t \\ 0 & \Delta t - C_2 \end{bmatrix} X[k] + \begin{bmatrix} 0 \\ C_1 \Delta t \end{bmatrix} V$$

this could be a Δt^2 term but in practice Δt^2 is tiny (1×10^{-6}) so we ignore here.

Some interesting notes:

$\ddot{\theta} = -C_2\dot{\theta} + C_1 V$, so what if we have constant velocity? ($\ddot{\theta} = 0$)

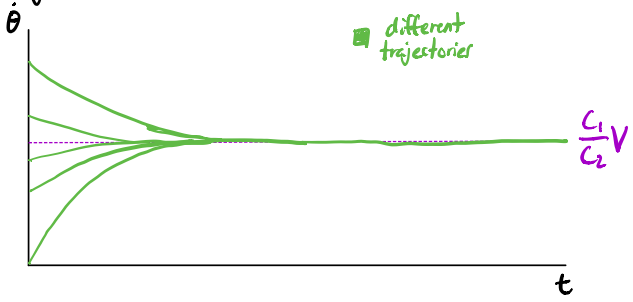
$0 = C_1 V - C_2\dot{\theta} \Rightarrow C_1 V = C_2\dot{\theta} \Rightarrow \dot{\theta} = \frac{C_1}{C_2} V$, so our final velocity is only a function of voltage i.e we want to go 500 radians/second so apply $C_2/C_1 (500)$ V.

If our voltage is fixed, i.e $\dot{V} = 0$ then $\ddot{\theta} = -C_2\dot{\theta} + C_1 V$, and we solve the differential equation to get:

$$\dot{\theta} = \frac{C_1}{C_2} V + De^{-C_2 t}$$

{ the sign of D depends on starting velocity.

All trajectories look like



GoogleTest Arm

Have arm_test.cc, arm.cc, and arm.hh open while reading!

To start we define a bunch of constants in our test fixture class ArmTest. These are used to calculate k_v and k_t for our motor. We also need a helper function to do the acceleration calculation, and a function to simulate forwards for some time. We choose simulation Δt to be 0.0001 seconds. This loop advances our position, and velocity. We can write a test around this using the TEST_F macro from Googletest like this test, of course, trivially passes.

```
TEST_F(ArmTest, Zeros) { }
```

Next, we need something to actually test, so we build the controller class ArmLoop. We define an entrypoint function:

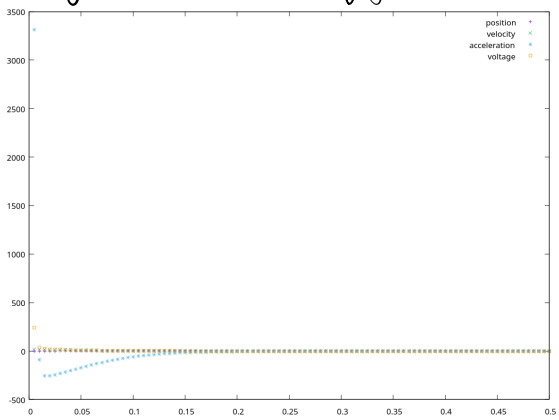
```
double update(double encoder, bool lower_limit, bool upper_limit, bool enabled);
```

measured position — lower hall on — upper hall on — is the robot enabled

We can for now just return 1.0 as the voltage and fix that later. Now we update the sim code to use this:

```
TEST_F(ArmTest, Zeros) {
  ArmLoop arm;  create an ArmLoop
  arm.set_goal(50 * M_PI / 180);  set the goal to 50°
  double current_time = 0.0;
  while (current_time < 1.0) {  run for 1 second
    double voltage = arm.update(  get the voltage to
        position_, lower_limit_triggered,  apply
        upper_limit_triggered, true);
    simulate_for_time(voltage, ArmLoop::DT);  run time forward by dT
    current_time += ArmLoop::DT;  step our tracking time forwards
  }
  EXPECT_NEAR(50 * M_PI / 180, position_, 0.01);  check if we got to the right position
}
```

Obviously, this test fails. Applying 1.0 V is not a good way to get to the right position.



Here is a graph of this run. Since we're applying constant voltage, we get exactly the velocity curve we expected earlier from the math!

Obviously we can do better.

We will do better than this, so we'll write a PD controller, which you've seen before.

That looks like this:

```
double error = goal_ - encoder;
double error_vel = last_error_ - error;
last_error_ = error;
return kP * error + kD * error_vel;
```

With decent tuned constants, we get much better results and pass our tests. We'll continue to build from here.

Looking at the graph on the right there are a few noticable problems we should also check in our test, i.e our test passes here but we wouldn't want this to pass: we are missing failure checks.