studywolf

a blog for things I encounter while coding and researching neuroscience, motor control, and learning

FEB 03 2016

29 COMMENTS
BY TRAVISDEWOLF LINEAR ALGEBRA, MATH, MOTOR CONTROL, PROGRAMMING, PYTHON, ROBOTICS

The iterative Linear Quadratic Regulator algorithm

A few months ago I posted on Linear Quadratic Regulators (LQRs) for control of non-linear systems using finite-differences (https://studywolf.wordpress.com/2015/11/10/linear-quadratic-regulation-for-non-linear-systems-using-finite-differences/). The gist of it was at every time step linearize the dynamics, quadratize (it could be a word) the cost function around the current point in state space and compute your feedback gain off of that, as though the dynamics were both linear and consistent (i.e. didn't change in different states). And that was pretty cool because you didn't need all the equations of motion and inertia matrices etc to generate a control signal. You could just use the simulation you had, sample it a bunch to estimate the dynamics and value function, and go off of that.

The LQR, however, operates with maverick disregard for changes in the future. Careless of the consequences, it optimizes assuming the linear dynamics approximated at the current time step hold for all time. It would be really great to have an algorithm that was able to plan out and optimize a sequence, mindful of the changing dynamics of the system.

This is exactly the iterative Linear Quadratic Regulator method (iLQR) was designed for. iLQR is an extension of LQR control, and the idea here is basically to optimize a whole control sequence rather than just the control signal for the current point in time. The basic flow of the algorithm is:

1. Initialize with initial state x_0 and initial control sequence $\mathbf{U} = [u_{t_0}, u_{t_1}, ..., u_{t_{N-1}}]$.
2. Do a forward pass, i.e. simulate the system using $(x_0, \mathbf{U})$ to get the trajectory through state space, $\mathbf{X}$, that results from applying the control sequence $\mathbf{U}$ starting in x_0.
3. Do a backward pass, estimate the value function and dynamics for each $(\mathbf{x}, \mathbf{u})$ in the state-space and control signal trajectories.
4. Calculate an updated control signal $\hat{\mathbf{U}}$ and evaluate cost of trajectory resulting from $(x_0, \hat{\mathbf{U}})$.
   1. If $|(\text{cost}(x_0, \hat{\mathbf{U}}) - \text{cost}(x_0, \mathbf{U})| < \text{threshold}$ then we've converged and exit.
   2. If $\text{cost}(x_0, \hat{\mathbf{U}}) < \text{cost}(x_0, \mathbf{U})$, then set $\mathbf{U} = \hat{\mathbf{U}}$, and change the update size to be more aggressive. Go back to step 2.

3. If $\text{cost}(x_0, \hat{\mathbf{U}}) \geq \text{cost}(x_0, \mathbf{U})$ change the update size to be more modest. Go back to step 3.

There are a bunch of descriptions of iLQR, and it also goes by names like 'the sequential linear quadratic algorithm'. The paper that I'm going to be working off of is by Yuval Tassa out of Emo Todorov's lab, called Control-limited differential dynamic programming (https://homes.cs.washington.edu/~todorov/papers/TassaICRA14.pdf). And the Python implementation of this can be found up on my github (https://github.com/studywolf/control/blob/master/studywolf_control/controllers/ilqr.py) in my Control repo. Also, a big thank you to Dr. Emo Todorov who provided Matlab code for the iLQG algorithm, which was super helpful.

Defining things

So let's dive in. Formally defining things, we have our system $\mathbf{x}$, and dynamics described with the function $\mathbf{f}$, such that

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t),$$

where $\mathbf{u}$ is the input control signal. The trajectory $\{\mathbf{X}, \mathbf{U}\}$ is the sequence of states $\mathbf{X} = \{\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_N\}$ that result from applying the control sequence $\mathbf{U} = \{\mathbf{u}_0, \mathbf{u}_1, ..., \mathbf{u}_{N-1}\}$ starting in the initial state $\mathbf{x}_0$.

Now we need to define all of our cost related equations, so we know exactly what we're dealing with.

Define the *total cost* function J, which is the sum of the immediate cost, ℓ, from each state in the trajectory plus the final cost, ℓ_f:

$$J(\mathbf{x}_0, \mathbf{U}) = \sum_{t=0}^{N-1} \ell(\mathbf{x}_t, \mathbf{u}_t) + \ell_f(\mathbf{x}_N).$$

Letting $\mathbf{U}_t = \{\mathbf{u}_t, \mathbf{u}_{t+1}, ..., \mathbf{U}_{N-1}\}$, we define the *cost-to-go* as the sum of costs from time t to N:

$$J_t(\mathbf{x}, \mathbf{U}_t) = \sum_{i=t}^{N-1} \ell(\mathbf{x}_i, \mathbf{u}_i) + \ell_f(\mathbf{x}_N).$$

The *value* function V at time t is the optimal cost-to-go from a given state:

$$V_t(\mathbf{x}) = \min_{\mathbf{U}_t} J_t(\mathbf{x}, \mathbf{U}_t),$$

where the above equation just says that the optimal cost-to-go is found by using the control sequence $\mathbf{U}_t$ that minimizes J_t.

At the final time step, N, the value function is simply

$$V(\mathbf{x}_N) = \ell_f(\mathbf{x}_N).$$

For all preceding time steps, we can write the value function as a function of the immediate cost $\ell(\mathbf{x}, \mathbf{u})$ and the value function at the next time step:

$$V(\mathbf{x}) = \min_{\mathbf{u}} \left[\ell(\mathbf{x}, \mathbf{u}) + V(\mathbf{f}(\mathbf{x}, \mathbf{u})) \right].$$

NOTE: In the paper, they use the notation $V'(\mathbf{f}(\mathbf{x}, \mathbf{u}))$ to denote the value function at the next time step, which is redundant since $\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t)$, but it comes in handy later when they drop the dependencies to simplify notation. So, heads up: $V' = V(\mathbf{f}(\mathbf{x}, \mathbf{u})$.

Forward rollout

The forward rollout consists of two parts. The first part is to simulating things to generate the $(\mathbf{X}, \mathbf{U}),$ from which we can calculate the overall cost of the trajectory, and find out the path that the arm will take. To improve things though we'll need a lot of information about the partial derivatives of the system, calculating these is the second part of the forward rollout phase.

To calculate all these partial derivatives we'll use $(\mathbf{X}, \mathbf{U})$. For each $(\mathbf{x}_t, \mathbf{u}_t)$ we'll calculate the derivatives of $\mathbf{f}(\mathbf{x}_t, \mathbf{u}_t)$ with respect to $\mathbf{x}_t$ and $\mathbf{u}_t$, which will give us what we need for our linear approximation of the system dynamics.

To get the information we need about the value function, we'll need the first and second derivatives of $\ell(\mathbf{x}_t, \mathbf{u}_t)$ and $\ell_f(\mathbf{x}_t, \mathbf{x}_t)$ with respect to $\mathbf{x}_t$ and $\mathbf{u}_t$.

So all in all, we need to calculate $\mathbf{f}_\mathbf{x}$, $\mathbf{f}_\mathbf{u}$, $\ell_\mathbf{x}$, $\ell_\mathbf{u}$, $\ell_\mathbf{xx}$, $\ell_\mathbf{ux}$, $\ell_\mathbf{uu}$, where the subscripts denote a partial derivative, so $\ell_\mathbf{x}$ is the partial derivative of ℓ with respect to $\mathbf{x}$, $\ell_\mathbf{xx}$ is the second derivative of ℓ with respect to $\mathbf{x}$, etc. And to calculate all of these partial derivatives, we're going to use finite differences! Just like in the LQR with finite differences post (https://studywolf.wordpress.com/2015/11/10/linear-quadratic-regulation-for-non-linear-systems-using-finite-differences/). Long story short, load up the simulation for every time step, slightly vary one of the parameters, and measure the resulting change.

Once we have all of these, we're ready to move on to the backward pass.

Backward pass

Now, we started out with an initial trajectory, but that was just a guess. We want our algorithm to take it and then converge to a local minimum. To do this, we're going to add some perturbing values and use them to minimize the value function. Specifically, we're going to compute a local solution to our value function using a quadratic Taylor expansion. So let's define $Q(\delta\mathbf{x}, \delta\mathbf{u})$ to be the change in our value function at $(\mathbf{x}, \mathbf{u})$ as a result of small perturbations $(\delta\mathbf{x}, \delta\mathbf{u})$:

$$Q(\delta\mathbf{x}, \delta\mathbf{u}) = \ell(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u}) + V(\mathbf{f}(\mathbf{x} + \delta\mathbf{x}, \mathbf{u} + \delta\mathbf{u})).$$

The second-order expansion of Q is given by:

$$Q_\mathbf{x} = \ell_\mathbf{x} + \mathbf{f}_\mathbf{x}^T V'_\mathbf{x},$$

$$Q_\mathbf{u} = \ell_\mathbf{u} + \mathbf{f}_\mathbf{u}^T V'_\mathbf{x},$$

$$Q_\mathbf{xx} = \ell_\mathbf{xx} + \mathbf{f}_\mathbf{x}^T V'_\mathbf{xx} \mathbf{f}_\mathbf{x} + V'_\mathbf{x} \cdot \mathbf{f}_\mathbf{xx},$$

$$Q_{\mathbf{ux}} = \ell_{\mathbf{ux}} + \mathbf{f}_{\mathbf{u}}^T V'_{\mathbf{xx}} \mathbf{f}_{\mathbf{x}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{ux}},$$

$$Q_{\mathbf{uu}} = \ell_{\mathbf{uu}} + \mathbf{f}_{\mathbf{u}}^T V'_{\mathbf{xx}} \mathbf{f}_{\mathbf{u}} + V'_{\mathbf{x}} \cdot \mathbf{f}_{\mathbf{uu}}.$$

Remember that $V' = V(\mathbf{f}(\mathbf{x}, \mathbf{u}))$, which is the value function at the next time step. NOTE: All of the second derivatives of $\mathbf{f}$ are zero in the systems we're controlling here, so when we calculate the second derivatives we don't need to worry about doing any tensor math, yay!

Given the second-order expansion of Q, we can to compute the optimal modification to the control signal, $\delta\mathbf{u}^*$. This control signal update has two parts, a feedforward term, $\mathbf{k}$, and a feedback term $\mathbf{K}\delta\mathbf{x}$. The optimal update is the $\delta\mathbf{u}$ that minimizes the cost of Q:

$$\delta\mathbf{u}^*(\delta\mathbf{x}) = \min_{\delta\mathbf{u}} Q(\delta\mathbf{x}, \delta\mathbf{u}) = \mathbf{k} + \mathbf{K}\delta\mathbf{x},$$

where $\mathbf{k} = -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{u}}$ and $\mathbf{K} = -Q_{\mathbf{uu}}^{-1} Q_{\mathbf{ux}}$.

Derivation can be found in this earlier paper (https://homes.cs.washington.edu/~todorov/papers/LiICINCO04.pdf) by Li and Todorov. By then substituting this policy into the expansion of Q we get a quadratic model of V. They do some mathamagics and come out with:

$$V_{\mathbf{x}} = Q_{\mathbf{x}} - \mathbf{K}^T Q_{\mathbf{uu}} \mathbf{k},$$

$$V_{\mathbf{xx}} = Q_{\mathbf{xx}} - \mathbf{K}^T Q_{\mathbf{uu}} \mathbf{K}.$$

So now we have all of the terms that we need, and they're defined in terms of the values at the *next* time step. We know the value of the value function at the final time step $V_N = \ell_f(\mathbf{x}_N)$, and so we'll simply plug this value in and work backwards in time recursively computing the partial derivatives of Q and V.

Calculate control signal update

Once those are all calculated, we can calculate the gain matrices, $\mathbf{k}$ and $\mathbf{K}$, for our control signal update. Huzzah! Now all that's left to do is evaluate this new trajectory. So we set up our system

$$\hat{\mathbf{x}}_0 = \mathbf{x}_0,$$

$$\hat{\mathbf{u}}_t = \mathbf{u}_t + \mathbf{k}_t + \mathbf{K}_t(\hat{\mathbf{x}}_t - \mathbf{x}_t),$$

$$\hat{\mathbf{x}}_{t+1} = \mathbf{f}(\hat{\mathbf{x}}_t, \hat{\mathbf{u}}_t),$$

and record the cost. Now if the cost of the new trajectory $(\hat{\mathbf{X}}, \hat{\mathbf{U}})$ is less than the cost of $(\mathbf{X}, \mathbf{U})$ then we set $\mathbf{U} = \hat{\mathbf{U}}$ and go do it all again! And when the cost from an update becomes less than a threshold value, call it done. In code this looks like:

```python
if costnew < cost:
  sim_new_trajectory = True

  if (abs(costnew - cost)/cost) < self.converge_thresh:
    break
```

Of course, another option we need to account for is when `costnew > cost`. What do we do in this case? Our control update hasn't worked, do we just exit?

The Levenberg-Marquardt heuristic
No! Phew.

The control signal update in iLQR is calculated in such a way that it can behave like Gauss-Newton optimization (which uses second-order derivative information) or like gradient descent (which only uses first-order derivative information). The is that if the updates are going well, then lets include curvature information in our update to help optimize things faster. If the updates aren't going well let's dial back towards gradient descent, stick to first-order derivative information and use smaller steps. This wizardry is known as the Levenberg-Marquardt heuristic (https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm). So how does it work?

Something we skimmed over in the iLQR description was that we need to calculate $Q_{\mathbf{uu}}^{-1}$ to get the $\mathbf{k}$ and $\mathbf{K}$ matrices. Instead of using `np.linalg.pinv` or somesuch, we're going to calculate the inverse ourselves after finding the eigenvalues and eigenvectors, so that we can regularize it. This will let us do a couple of things. First, we'll be able to make sure that our estimate of curvature ($Q_{\mathbf{uu}}^{-1}$) stays positive definite, which is important to make sure that we always have a descent direction (http://www.math.mtu.edu/~msgocken/ma5630spring2003/lectures/global1/global1/node2.html). Second, we're going to add a regularization term to the eigenvalues to prevent them from exploding when we take their inverse. Here's our regularization implemented in Python:

```python
Q_uu_evals, Q_uu_evecs = np.linalg.eig(Q_uu)
Q_uu_evals[Q_uu_evals < 0] = 0.0
Q_uu_evals += lamb
Q_uu_inv = np.dot(Q_uu_evecs,
    np.dot(np.diag(1.0/Q_uu_evals), Q_uu_evecs.T))
```

Now, what happens when we change `lamb`? The eigenvalues represent the magnitude of each of the eigenvectors, and by taking their reciprocal we flip the contributions of the vectors. So the ones that were contributing the least now have the largest singular values, and the ones that contributed the most now have the smallest eigenvalues. By adding a regularization term we ensure that the inverted eigenvalues can never be larger than `1/lamb`. So essentially we throw out information.

In the case where we've got a really good approximation of the system dynamics and value function, we don't want to do this. We want to use all of the information available because it's accurate, so make `lamb` small and get a more accurate inverse. In the case where we have a bad approximation of the dynamics we want to be more conservative, which means not having those large singular values. Smaller singular values give a smaller $Q_{\mathbf{uu}}^{-1}$ estimate, which then gives smaller gain matrices and control signal update, which is what we want to do when our control signal updates are going poorly.

How do you know if they're going poorly or not, you now surely ask! Clever as always, we're going to use the result of the previous iteration to update `lamb`. So adding to the code from just above, the end of our control update loop is going to look like:

```python
lamb = 1.0 # initial value of lambda
...
if costnew < cost:
  lamb /= self.lamb_factor
  sim_new_trajectory = True

  if (abs(costnew - cost)/cost) < self.converge_thresh:
    break
else:
  lamb *= self.lamb_factor
  if lamb > self.max_lamb:
    break
```

And that is pretty much everything! OK let's see how this runs!

Simulation results

If you want to run this and see for yourself, you can go copy my Control repo (https://github.com/studywolf/control), navigate to the main directory, and run

```
python run.py arm2 reach
```

or substitute in `arm3`. If you're having trouble getting the `arm2` simulation to run, try `arm2_python`, which is a straight Python implementation of the arm dynamics, and should work no sweat for Windows and Mac.

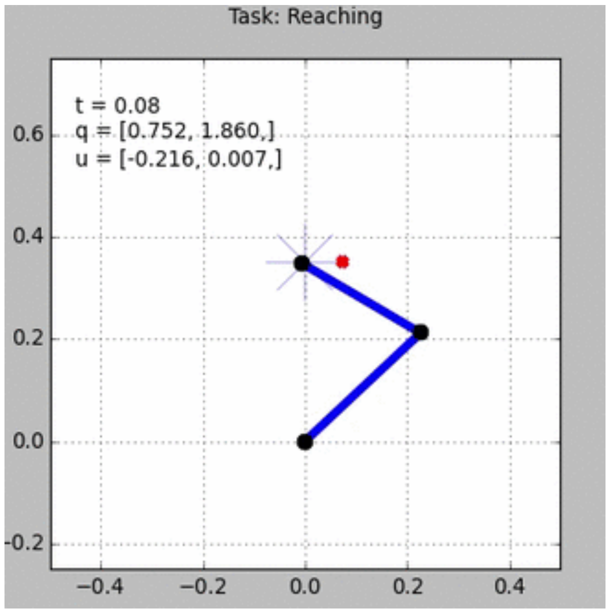
Below you can see results from the iLQR controller controlling the 2 and 3 link arms (click on the figures to see full sized versions, they got distorted a bit in the shrinking to fit on the page), using immediate and final state cost functions defined as:

```python
l = np.sum(u**2)
```

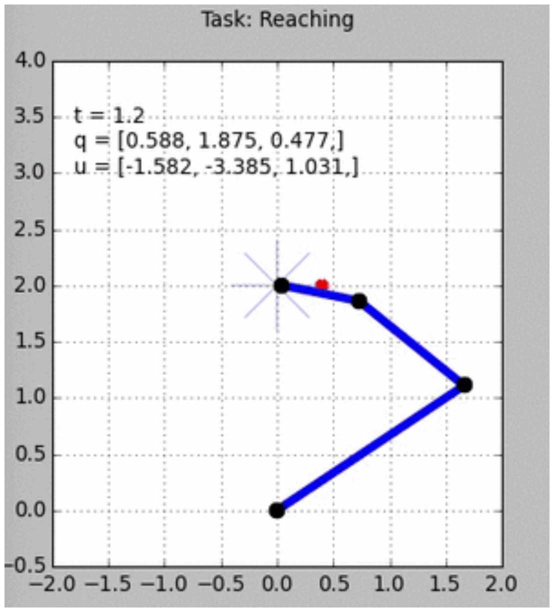
and

```python
pos_err = np.array([self.arm.x[0] - self.target[0],
                    self.arm.x[1] - self.target[1]])
l = (wp * np.sum(pos_err**2) + # pos error
    wv * np.sum(x[self.arm.DOF:self.arm.DOF*2]**2)) # vel error
```

where `wp` and `wv` are just gain values, `x` is the state of the system, and `self.arm.x` is the (x, y) position of the hand. These read as "during movement, penalize large control signals, and at the final state, have a big penalty on not being at the target."



(https://studywolf.wordpress.com/2016/02/03/the-iterative-linear-quadratic-regulator-method/2link/)

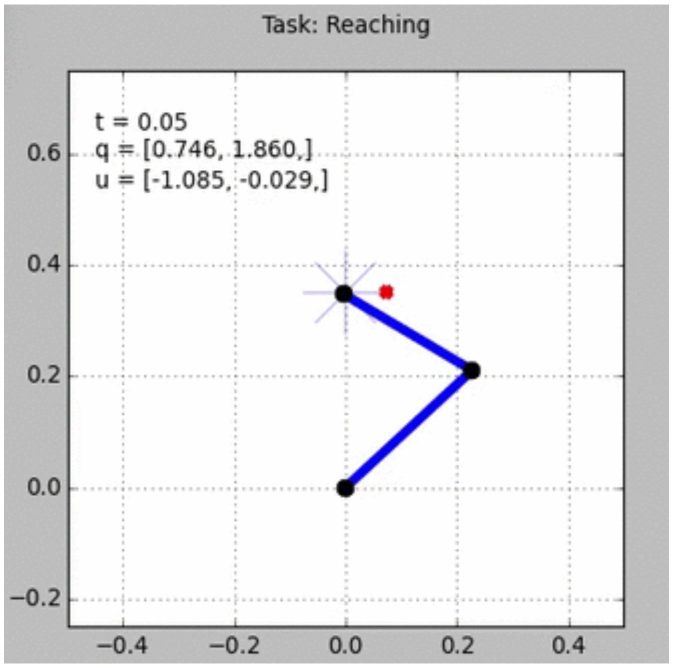


(https://studywolf.wordpress.com/2016/02/03/the-iterative-linear-quadratic-regulator-method/3link-2/)

So let's give it up for iLQR, this is awesome! How much of a crazy improvement is that over LQR? And with all knowledge of the system through finite differences, and with the full movements in exactly 1 second! (Note: The simulation speeds look different because of my editing to keep the gif sizes small, they both take the same amount of time for each movement.)

<u>Changing cost functions</u>
Something that you may notice is that the control of the 3 link is actually straighter than the 2 link. I thought that this might be just an issue with the gain values, since the scale of movement is smaller for the 2 link arm than the 3 link there might have been less of a penalty for not moving in a straight line, BUT this was wrong. You can crank the gains and still get the same movement. The actual reason is that this is what the cost function specifies, if you look in the code, only $\ell_f(\mathbf{x}_N)$ penalizes the distance from the target, and the cost function during movement is strictly to minimize the control signal, i.e. $\ell(\mathbf{x}_t, \mathbf{u}_t) = \mathbf{u}_t^2$.

Well that's a lot of talk, you say, like the incorrigible antagonist we both know you to be, prove it. Alright, fine! Here's iLQR running with an updated cost function that includes the end-effector's distance from the target in the immediate cost:



All that I had to do to get this was change the immediate cost from

```
l = np.sum(u**2)
```

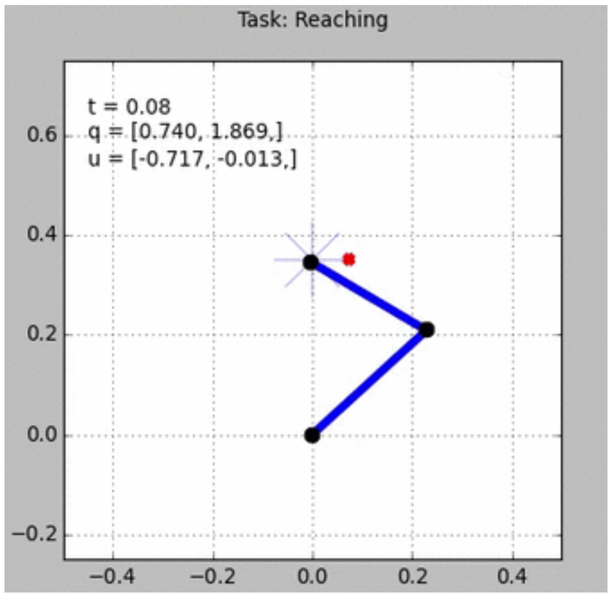
to

```
l = np.sum(u**2)
pos_err = np.array([self.arm.x[0] - self.target[0],
                    self.arm.x[1] - self.target[1]])
l += (wp * np.sum(pos_err**2) + # pos error
    wv * np.sum(x[self.arm.DOF:self.arm.DOF*2]**2)) # vel error
```

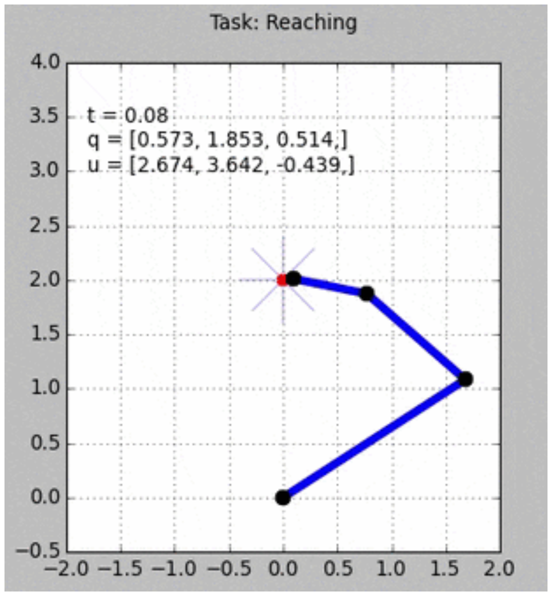
where all I had to do was include the position penalty term from the final state cost into the immediate state cost.

<u>Changing sequence length</u>

In these simulations the system is simulating at .01 time step, and I gave it 100 time steps to reach the target. What if I give it only 50 time steps?



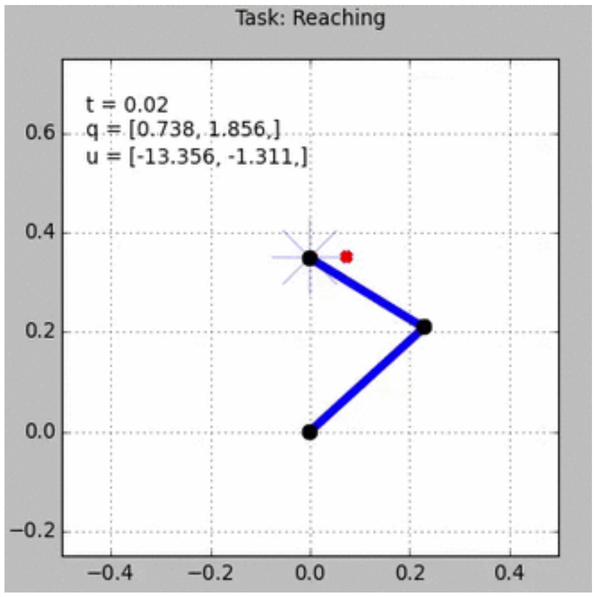
(https://studywolf.wordpress.com/2016/02/03/the-iterative-linear-quadratic-regulator-method/2link50/)



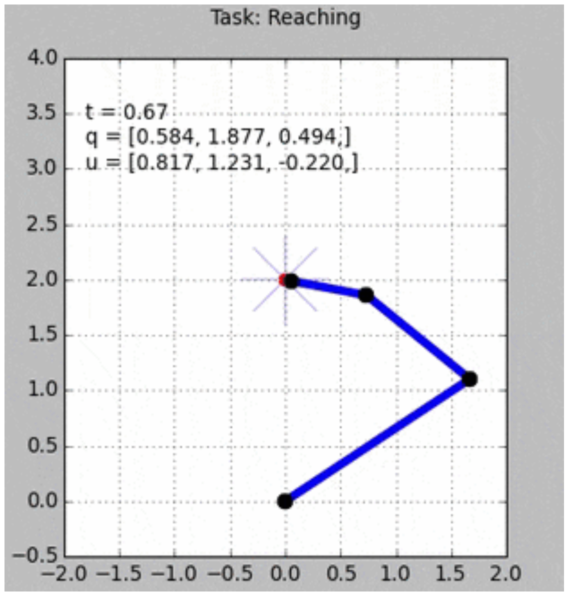
(https://studywolf.wordpress.com/2016/02/03/the-iterative-linear-quadratic-regulator-method/3link50/)

It looks pretty much the same! It's just now twice as fast, which is of course achieved by using larger control signals, which we don't see, but dang awesome.

What if we try to make it there in 10 time steps??



(https://studywolf.wordpress.com/2016/02/03/the-iterative-linear-quadratic-regulator-method/2link10/)



(https://studywolf.wordpress.com/2016/02/03/the-iterative-linear-quadratic-regulator-method/3link10/)

OK well that does not look good. So what's going on in this case? Basically we've given the algorithm an impossible task. It can't make it to the target location in 10 time steps. In the implementation I wrote here, if it hits the end of it's control sequence and it hasn't reached the target yet, the control sequence starts over back at `t=0`. Remember that part of the target state is also velocity, so basically it moves for 10 time steps to try to minimize (x, y) distance, and then slows down to minimize final state cost in the velocity term.

In conclusion

This algorithm has been used in a ton of things, for controlling robots and simulations, and is an important part of guided policy search (https://graphics.stanford.edu/projects/gpspaper/gps_full.pdf), which has been used to very successfully train deep networks in control problems. It's getting really impressive results for controlling the arm models that I've built here, and using finite differences should easily generalize to other systems.

iLQR is very computationally expensive, though, so that's definitely a downside. It's definitely less expensive if you have the equations of your system, or at least a decent approximation of them, and you don't need to use finite differences. But you pay for the efficiency with a loss in generality.

There are also a bunch of parameters to play around with that I haven't explored at all here, like the weights in the cost function penalizing the magnitude of the cost function and the final state position error. I showed a basic example of changing the cost function, which hopefully gets across just how easy changing these things out can be when you're using finite differences, and there's a lot to play around with there too.

<u>Implementation note</u>
In the Yuval and Todorov paper, they talked about using backtracking line search when generating the control signal. So the algorithm they had when generating the new control signal was actually:

$$\hat{\mathbf{u}}_t = \hat{\mathbf{u}}_t + \alpha \mathbf{k}_t + \mathbf{K}_t(\hat{\mathbf{x}}_t - \mathbf{x}_t)$$

where α was the backtracking search parameter, which gets set to one initially and then reduced. It's very possible I didn't implement it as intended, but I found consistently that $\alpha = 1$ always generated the best results, so it was just adding computation time. So I left it out of my implementation. If anyone has insights on an implementation that improves results, please let me know!

And then finally, another thank you to Dr. Emo Todorov for providing Matlab code for the iLQG algorithm, which was very helpful, especially for getting the Levenberg-Marquardt heuristic implemented properly.

Tagged arm control, control, control theory, iterative linear quadratic regulator, motor control, programming, Python, robot control, robotics