Underactuated Robots
Lecture 3: Differential Dynamic Programming

Nicola Scianca

September 2024

introduction

- the Bellman equation is at the core of **dynamic programming**, a technique for solving optimization problems by breaking them down into smaller subproblems

- it defines a **recursive relationship**, where the expected value at a particular time depends on the value at a future time

- it has important applications not only in optimal control and trajectory optimization, but also in reinforcement learning and other areas

the value function

$$\boxed{\begin{array}{c} V_k(x_k) = \min_{u_k,\ldots,u_{N-1}} \left(\sum_{i=k}^{N-1} l_k(x_i, u_i) + l_N(x_N) \right) \\ \text{s. t.} \qquad x_{k+1} = f(x_k, u_k) \end{array}}$$

- the **value function** $V_k(x_k)$ tells us the cost we will pay if we always take optimal actions in the future
- the cost is a reward in maximization problems, hence the name "value function"
- at time k, it depends on the state x_k, but not on the input (we are choosing the best possible input sequence)

the Bellman equation

- now, take out the first item of the sum

$$\begin{aligned} V_k(x_k) &= \min_{u_k,\dots,u_{N-1}} \left(l_k(x_k,u_k) + \sum_{i=k+1}^{N-1} l_k(x_i,u_i) + l_N(x_N) \right) \\ &= \min_{u_k} \left(l_k(x_k,u_k) + \boxed{\min_{u_{k+1},\dots,u_{N-1}} \sum_{i=k+1}^{N-1} l_k(x_i,u_i) + l_N(x_N)} \right) \\ &= \min_{u_k} \left(l_k(x_k,u_k) + V_{k+1}(x_{k+1}) \right) \end{aligned}$$

- since $\boxed{}$ is the value function at $k+1$, we get a nice recursive expression: this is the Bellman equation

the Bellman equation

- remember: this minimization must take into account the dynamic constraint $x_{k+1} = f(x_k, u_k)$

- if we substitute this inside the recursive expression, we get

$$\boxed{V(x_k) = \min_{u_k} \left(l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k)) \right)}$$

which is the Bellman equation in the discrete case

the Bellman equation

$$\boxed{V(x_k) = \min_{u_k} \left(l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k)) \right)}$$

- the Bellman equation relates the value of V_k to the value of V_{k+1} in a recursive way
- if we had perfect knowledge of V, this would make it easy to derive an optimal control law, but usually we cannot explicitly solve for V and we have to approximate
- however, there is at least one case in which we can find an explicit solution: the **Linear Quadratic Regulator** (LQR)

linear quadratic regulator

- in the LQR we are trying to regulate the state x to zero, thus the cost function to minimize looks like

$$J = \sum_{k=0}^{N-1} \left(x_k^T Q x_k + u_k^T R u_k \right) + x_N^T Q_f x_N$$

where $Q \succeq 0$, $R \succ 0$, and $Q_f \succeq 0$ are weight matrices

- the system dynamics equation is linear

$$x_{k+1} = A x_k + B u_k$$

linear quadratic regulator

- the value function at time step k is

$$V_k(x_k) = \min_{u_k,\dots,u_{N-1}} \sum_{i=k}^{N-1} \left(x_i^T Q x_i + u_i^T R u_i\right) + x_N^T Q_f x_N$$

- using the Bellman equation we can write it recursively as

$$V_k(x_k) = \min_{u_k} \left(x_k^T Q x_k + u_k^T R u_k + V_{k+1}(x_{k+1})\right)$$

with system dynamics $x_{k+1} = Ax_k + Bu_k$.

linear quadratic regulator

- let's assume that the value function is **quadratic**

$$V_k(x_k) = x_k^T P_k x_k$$

- the value function at the next time-step $k+1$ can be related to the current x_k and u_k via the system dynamics

$$V_{k+1} = x_{k+1}^T P_{k+1} x_{k+1} = (Ax_k + Bu_k)^T P_{k+1} (Ax_k + Bu_k)$$

- substitute into the Bellman equation:

$$V_k(x_k) = \min_{u_k} \Big(x_k^T Q x_k + u_k^T R u_k \\ + (Ax_k + Bu_k)^T P_{k+1} (Ax_k + Bu_k) \Big)$$

linear quadratic regulator

- to compute the minimum, derive with respect to u_k and set equal to zero

$$\frac{d}{du_k}\left(x_k^T Q x_k + u_k^T R u_k + (Ax_k + Bu_k)^T P_{k+1}(Ax_k + Bu_k)\right)$$
$$= \cancel{2}Ru_k + \cancel{2}B^T P_{k+1}(Ax_k + Bu_k)$$
$$= (R + B^T P_{k+1} B)u_k + B^T P_{k+1} A x_k = 0$$
$$\implies \quad u_k = -(R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A x_k$$

- substituting into the Bellman equation yields the **Riccati recursion**

$$P_k = Q + A^T P_{k+1} A - A^T P_{k+1} B (R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$

linear quadratic regulator

- now we start from the final state with $P_N = Q_f$, and going backwards with the Riccati recursion we can compute P_k at each time step

$$P_k = Q + A^T P_{k+1} A - A^T P_{k+1} B (R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$

- once we have full knowledge of P_k (which means full knowledge of the value function), the optimal control law is simply given by

$$u_k = -(R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A x_k$$

differential dynamic programming

- if the systems dynamics are nonlinear, we usually can't directly solve for the value function
- if we discretize the state space we can compute V numerically everywhere (dynamic programming), but for large systems this is impossible due to the **curse of dimensionality**
- what we can do is find a **quadratic approximation** of the value function: **Differential Dynamic Programming** (DDP)

quadratic approximation of the value function

- let's recall the Bellman equation

$$V(x_k) = \min_{u_k} \left(l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k)) \right)$$

- call Q the argument of the minimization

$$Q(x_k, u_k) = l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k))$$

quadratic approximation of the value function

- a quadratic approximation Q, around the point $(\bar{x}_k, \bar{u}_k)$, would look like

$$Q(x_k, u_k) \simeq Q(\bar{x}_k, \bar{u}_k) + \left.\frac{dQ}{dx_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k + \left.\frac{dQ}{du_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta u_k$$
$$+ \frac{1}{2}\Delta x_k^T \left.\frac{d^2Q}{dx_k^2}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k + \frac{1}{2}\Delta u_k^T \left.\frac{d^2Q}{du_k^2}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta u_k$$
$$+ \Delta u_k^T \left.\frac{d^2Q}{dx_k du_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k$$

- let's use a more compact notation

$$\tilde{Q}(x_k, u_k) = \bar{Q}_k + \bar{Q}_k^x \Delta x_k + \bar{Q}_k^u \Delta u_k$$
$$+ \frac{1}{2}\Delta x_k^T \bar{Q}_k^{xx} \Delta x_k + \frac{1}{2}\Delta u_k^T \bar{Q}_k^{uu} \Delta u_k + \Delta u_k^T \bar{Q}_k^{ux} \Delta x_k^T$$

quadratic approximation of the value function

- let's compute, as an example, $\bar{Q}_k^x$

$$\bar{Q}_k^x = \frac{\partial}{\partial x_k}\Big(l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k))\Big) = \left.\frac{\partial l}{\partial x_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} + \left.\frac{\partial V_{k+1}}{\partial x_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}}$$

- since V_{k+1} is the value function at $k+1$, it doesn't depend on x_k directly but through the dynamics $f(x_k, u_k)$; therefore, we must use the chain rule

$$\bar{Q}_k^x = \left.\frac{\partial l}{\partial x_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} + \left.\frac{\partial V_{k+1}}{\partial x_{k+1}}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \left.\frac{\partial f}{\partial x_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} = \bar{l}_k^x + V_{k+1}^x \bar{f}_k^x$$

quadratic approximation of the value function

- similarly we can compute all the other terms

$$
\begin{aligned}
\bar{Q}_k^x &= l_k^x + V_{k+1}^x f_k^x \\
\bar{Q}_k^u &= l_k^u + V_{k+1}^x f_k^u \\
\bar{Q}_k^{xx} &= l_k^{xx} + (f_k^x)^T V_{k+1}^{xx} f_k^x + \cancel{V_{k+1}^x f_k^{xx}} \\
\bar{Q}_k^{uu} &= l_k^{uu} + (f_k^u)^T V_{k+1}^{xx} f_k^u + \cancel{V_{k+1}^x f_k^{uu}} \\
\bar{Q}_k^{ux} &= l_k^{ux} + (f_k^u)^T V_{k+1}^{xx} f_k^x + \underbrace{\cancel{V_{k+1}^x f_k^{ux}}}_{\text{second derivative, usually neglected}}
\end{aligned}
$$

- the second-order derivatives of the dynamics are often neglected because they take a lot of time to compute (this variant of DDP is sometimes called iLQR or iLQG)

quadratic approximation of the value function

- remember that Q is the argument of the minimization; to find the value function approximation we derive it with respect to Δu_k and set it equal to zero

$$\frac{\partial \tilde{Q}(x_k, u_k)}{\partial \Delta u_k} = \bar{Q}_k^u + \bar{Q}_k^{uu} \Delta u_k + \bar{Q}_k^{ux} \Delta x_k = 0$$

- the optimal Δu_k is therefore

$$\Delta u_k = -(\bar{Q}_k^{uu})^{-1} \left(\bar{Q}_k^u + \bar{Q}_k^{ux} \Delta x_k \right)$$

- this is a **local feedback** control law, which we can write as

$$\Delta u_k = k_k + K_k \Delta x_k \qquad \begin{array}{l} k_k = -(\bar{Q}_k^{uu})^{-1} \bar{Q}_k^u \\ K_k = -(\bar{Q}_k^{uu})^{-1} \bar{Q}_k^{ux} \end{array}$$

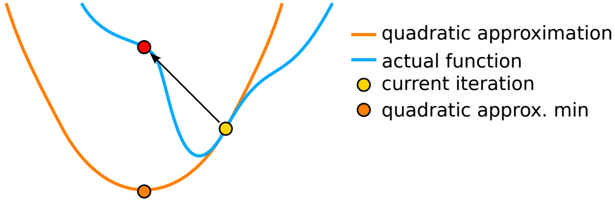
k_k is a feedforward term, and K_k is a local optimal gain

DDP algorithm

- start with an initial guess trajectory $(\bar{x}, \bar{u})$
- backward pass:
  - compute the value function approximation in the terminal state x_N
  - compute the approximation in the previous state x_{N-1}
  - in the process, you also obtain the optimal control law (k_k, K_k)
  - repeat until reaching the start of the horizon
- forward pass:
  - start from the current state $x_0 = \bar{x}_0$
  - compute the new input using the local gains $u_k = \bar{u}_k + k_k + K_k \Delta x_k$
  - propagate the dynamics forward $x_{k+1} = f(x_k, u_k)$
  - repeat until reaching the end of the horizon
- we now have a new guess trajectory $(\bar{x}, \bar{u}) \leftarrow (x, u)$
- repeat backward pass and forward pass until convergence

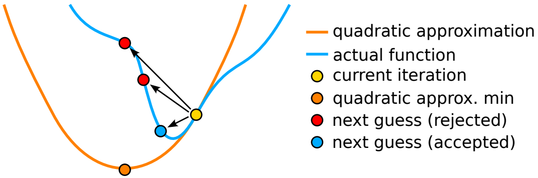
line search

- the local optimal control law $\Delta u_k = k_k + K_k \Delta x_k$ is only valid for a quadratic approximation, not for the true value function
- in reality, the value function is not quadratic, so this adjustment might overshoot the local minimum, and increase the cost of the new guess instead of decreasing it



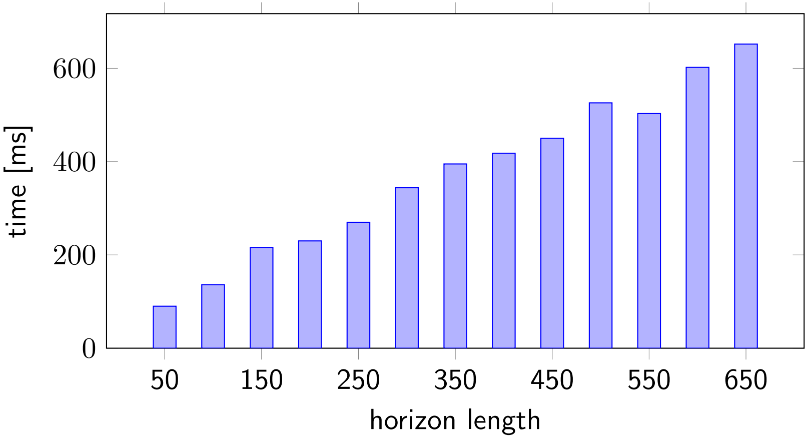
line search

- therefore, we modify the forward pass and perform something similar to a **line search** procedure:
  - start with an $\alpha = 1$
  - propagate the dynamics $x_{k+1} = f(x_k, \bar{u}_k + \alpha k_k + K_k \Delta x_k)$
  - evaluate the cost function over the new trajectory
  - if it increased, choose a smaller α and repeat the forward pass
  - if it decreased, accept the new trajectory (line search finished)



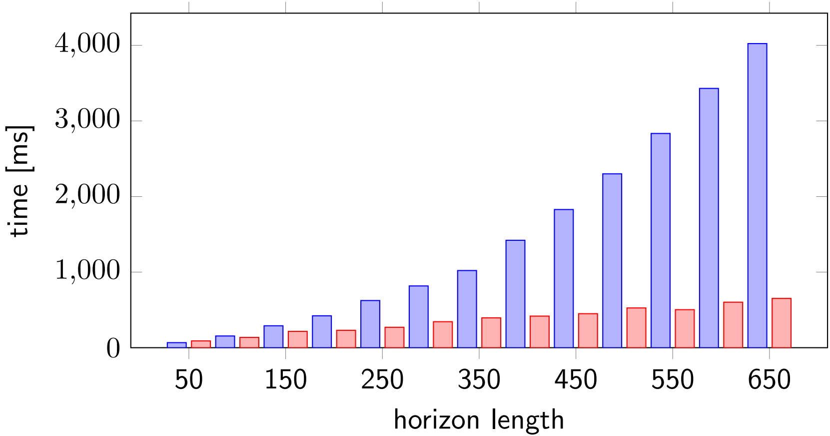
computational complexity

- no large matrix factorization is required! DDP scales **linearly** with the number of variables



computational complexity

- compared with the way SQP scales, we have a clear advantage when operating with a long horizon



example

```python
import ... [numpy, casadi, model, etc ...]

# parameters
n, m = 4, 1
N = 100
max_ddp_iters = 10
max_line_search_iters = 10
Q = np.eye(n) * 0
R = np.eye(m) * 0.01
Q_ter = np.eye(n) * 10000
x_ter = np.array((math.pi, 0, 0, 0))

# symbolic variables
opt = cs.Opti()
X = opt.variable(n)
U = opt.variable(m)

# ... continues in the next slide ->
```

example

```python
# cost function
L_ = lambda x, u: (x_ter - x).T @ Q @ (x_ter - x) + u.T @ R @ u
L_ter_ = lambda x: (x_ter - x).T @ Q_ter @ (x_ter - x)
L       = cs.Function('L'       , [X, U], [L_(X,U)])
L_ter   = cs.Function('L_ter'   , [X]   , [L_ter_(X)])
Lx      = cs.Function('Lx'      , [X, U], [cs.jacobian(L(X,U), X)])
Lu      = cs.Function('Lu'      , [X, U], [cs.jacobian(L(X,U), U)])
Lxx     = cs.Function('Lxx'     , [X, U], [cs.jacobian(Lx(X,U), X)])
Lux     = cs.Function('Lux'     , [X, U], [cs.jacobian(Lu(X,U), X)])
Luu     = cs.Function('Luu'     , [X, U], [cs.jacobian(Lu(X,U), U)])
L_terx  = cs.Function('L_terx'  , [X]   , [cs.jacobian(L_ter(X), X)])
L_terxx = cs.Function('L_terxx', [X]    , [cs.jacobian(L_terx(X), X)])

# dynamics
f_ = model.get_pendubot_model()
f  = cs.Function('f' , [X, U], [f_(X,U)])
fx = cs.Function('fx', [X, U], [cs.jacobian(f_(X,U), X)])
fu = cs.Function('fu', [X, U], [cs.jacobian(f_(X,U), U)])

# ... continues in the next slide ->
```

example

```python
for iter in range(max_ddp_iters):
  # backward pass
  backward_pass_start_time = time.time()
  V[N] = L_ter(x[:,N])
  Vx[:,N] = L_terx(x[:,N])
  Vxx[:,:,N] = L_terxx(x[:,N])

  for i in reversed(range(N)):
    fx_eval = fx(x[:,i], u[:,i])
    fu_eval = fu(x[:,i], u[:,i])

    Qx = Lx(x[:,i], u[:,i]).T + fx_eval.T @ Vx[:,i+1]
    Qu = Lu(x[:,i], u[:,i]).T + fu_eval.T @ Vx[:,i+1]

    Qxx = Lxx(x[:,i], u[:,i]) + fx_eval.T @ Vxx[:,:,i+1] @ fx_eval
    Quu = Luu(x[:,i], u[:,i]) + fu_eval.T @ Vxx[:,:,i+1] @ fu_eval
    Qux = Lux(x[:,i], u[:,i]) + fu_eval.T @ Vxx[:,:,i+1] @ fx_eval

    Quu_inv = np.linalg.inv(Quu)
    k[i] = - Quu_inv @ Qu
    K[i] = - Quu_inv @ Qux

    V[i] = V[i+1] - 0.5 * k[i].T @ Quu @ k[i]
    Vx[:,i] = np.array(Qx - K[i].T @ Quu @ k[i]).flatten()
    Vxx[:,:,i] = Qxx - K[i].T @ Quu @ K[i]

    # ... continues in the next slide ->
```

example

```python
# forward pass
forward_pass_start_time = time.time()
unew = np.ones((m, N))
xnew = np.zeros((n, N+1))
xnew[:,0] = x[:,0]

# line search
alpha = 1.
for ls_iter in range(max_line_search_iters):
  new_cost = 0
  for i in range(N):
    unew[:,i] = u[:,i] + alpha * k[i] + K[i] @ (xnew[:,i] - x[:,i])
    xnew[:,i+1] = np.array(f(xnew[:,i], unew[:,i])).flatten()
    new_cost = new_cost + L(xnew[:,i], unew[:,i])
  new_cost = new_cost + L_ter(xnew[:,N])

  if new_cost < cost:
    cost = new_cost
    x = xnew
    u = unew
    break
  else:
    alpha /= 2.
```

pros and cons

- pro: the computations are propagated forward and backward along the horizon: if you double the horizon length you just double the number of computations
- this means that the computational cost is **linear** with the horizon length
- by contrast, the complexity of SQP depends on the complexity of the QP subproblem, which in turns depends on the solver used (we don't know it apriori)

pros and cons

- con: it does not directly allow to set **constraints**, whereas in SQP this is straightforward

- con: it is a **single shooting** method, which means that when integrating the initial input guess you could get a diverging trajectory

- pro: there are variants of DDP that use an initial guess on the state, incorporating defects in a way that is similar to **multiple shooting** [Mastalli et al., 2020]

Mastalli et al.,"Crocoddyl: An Efficient and Versatile Framework for Multi-Contact Optimal Control", Robotics and Automation Letters 2020