Underactuated Robots
Lecture 2: Sequential Quadratic Programming

Nicola Scianca

September 2024

introduction

- **Sequential Quadratic Programming** (SQP) is an iterative method for nonlinear optimization, solving a sequence of optimization subproblems
- at each iteration, SQP solves a **Quadratic Programming** (QP) approximation of the original problem, where the objective function is quadratic, and the constraints are linearized
- despite its simplicity, SQP can be quite effective; for example it is used to control the **MIT Humanoid** [Ding et al., 2023]

Ding et al.,"Implementation of Convex Optimization Control on the MIT Humanoid", Humanoid Whole-body Control Workshop, IROS 2023

sequential quadratic programming

- we want to solve a **NonLinear Program** (NLP) with the following structure

$$
\begin{aligned}
\min_{\substack{x_0,\dots,x_N \\ u_0,\dots,u_{N-1}}} \sum_{i=0}^{N-1} l_k(x_i, u_i) + l_N(x_N) \\
\text{s. t.} \quad x_{k+1} &= f(x_k, u_k) \quad \text{for} \quad k = 0, \dots, N-1 \\
x_0 &= \bar{x}_0
\end{aligned}
$$

outline of the algorithm

- start from an **initial guess** for the trajectory $(\bar{x}, \bar{u})$
- find a **linear-quadratic approximation** of the NLP
- this gives as a QP, if we solve it we find a **descent direction**
- do a **line search** to find a step size in this direction
- find a new guess $(\bar{x}, \bar{u}) \leftarrow (x, u)$, **repeat** from step 2

linear-quadratic approximation

- we want to find approximate expressions for the cost function and constraints

- first, we define Δx_k and Δu_k as the variations of the k-th state and input with respect to some solution guess $\bar{x}_k$ and $\bar{u}_k$

$$\Delta x_k = x_k - \bar{x}_k$$
$$\Delta u_k = u_k - \bar{u}_k$$

- by computing Δx_k and Δu_k we can update our guess, and iterate to get closer and closer to the optimal solution

linear-quadratic approximation

- we first find a quadratic approximation of the cost function

$$l(x_k, u_k) \simeq l(\bar{x}_k, \bar{u}_k) + \left.\frac{dl}{dx_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k + \left.\frac{dl}{du_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta u_k$$

$$+ \frac{1}{2}\Delta x_k^T \left.\frac{d^2 l}{dx_k^2}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k^T + \frac{1}{2}\Delta u_k^T \left.\frac{d^2 l}{du_k^2}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta u_k^T$$

$$+ \Delta u_k^T \left.\frac{d^2 l}{dx_k du_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k^T$$

$$l_N(x_N) \simeq l_N(\bar{x}_k) + \left.\frac{dl_N}{dx_k}\right|_{\bar{x}_k} \Delta x_k + \frac{1}{2}\Delta x_k^T \left.\frac{d^2 l_N}{dx_k^2}\right|_{\bar{x}_k} \Delta x_k^T$$

- and a linear approximation of the constraints

$$\bar{x}_{k+1} + \Delta x_{k+1} \simeq f(\bar{x}_k, \bar{u}_k) + \left.\frac{df}{dx_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k + \left.\frac{df}{du_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta u_k$$

linear-quadratic approximation

- let's write them in more compact form

$$
\begin{aligned}
l(x_k, u_k) &\simeq \bar{l}_k + \bar{l}_k^x \Delta x_k + \bar{l}_k^u \Delta u_k \\
&\quad + \frac{1}{2} \Delta x_k^T \bar{l}_k^{xx} \Delta x_k + \frac{1}{2} \Delta u_k^T \bar{l}_k^{uu} \Delta u_k + \Delta u_k^T \bar{l}_k^{ux} \Delta x_k \\
l_N(x_N) &\simeq \bar{l}_N + \bar{l}_N^x \Delta x_k + \frac{1}{2} \Delta x_k^T \bar{l}_N^{xx} \Delta x_k \\
f(x_k, u_k) &\simeq \bar{f}_k + \bar{f}_k^x \Delta x_k + \bar{f}_k^u \Delta u_k
\end{aligned}
$$

the quadratic subproblem

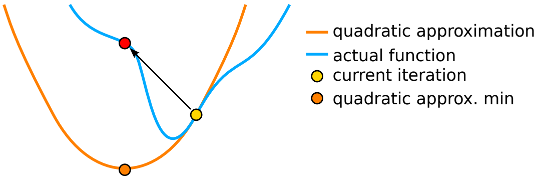
- now we set up a **quadratic subproblem**, approximating the original nonlinear problem around the current guess $(\bar{x}, \bar{u})$

$$
\begin{aligned}
\min_{\substack{\Delta x_0,\dots,\Delta x_N,\\ \Delta u_0,\dots,\Delta u_{N-1}}} \sum_{k=0}^{N} \Bigg(& \bar{l}_k^x \Delta x_k + \bar{l}_k^u \Delta u_k + \frac{1}{2} \Delta x_k^T \bar{l}_k^{xx} \Delta x_k \\
& + \frac{1}{2} \Delta u_k^T \bar{l}_k^{uu} \Delta u_k + \Delta u_k^T \bar{l}_k^{ux} \Delta x_k \Bigg) \\
& + \bar{l}_N + \bar{l}_N^x \Delta x_k + \frac{1}{2} \Delta x_k^T \bar{l}_N^{xx} \Delta x_k
\end{aligned}
$$

$$
\begin{aligned}
\text{s.t.} \quad \bar{x}_{k+1} + \Delta x_{k+1} &= \bar{f}_k + \bar{f}_k^x \Delta x_k + \bar{f}_k^u \Delta u_k \\
\Delta x_0 &= 0
\end{aligned}
$$

line search

- the solution to the previous problem represents a local **descent direction** for the NLP
- taking a full step $(\Delta x_k, \Delta u_k)$ in this direction takes us to the minimum of the local approximation, which might overshoot the true local minimum and increase the cost instead of decreasing it

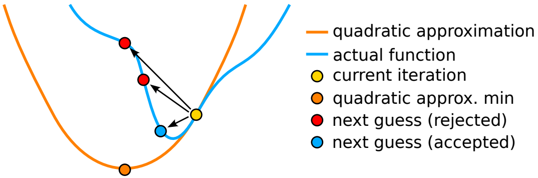


line search

- instead of taking a full step, we perform a **line search**:
  - start with an $\alpha = 1$
  - take a step of size α in the descent direction

$$\bar{x} \leftarrow \bar{x} + \alpha \Delta x$$
$$\bar{u} \leftarrow \bar{u} + \alpha \Delta u$$

  - evaluate the cost function on the new trajectory
  - if the cost is increased, choose a smaller α and retry
  - if the cost is decreased, accept the step (line search finished)



merit functions

- a point on the descent direction satisfies the linearized constraints, not necessarily the constraints of the NLP
- if we evaluate a line search step by looking at the reduction of the cost, we might end up **increasing constraint violation**
- therefore, it is sometimes useful to add a term that evaluates violation of the nonlinear constraints during the line search

$$M = \underbrace{C}_{\text{cost function}} + \sum_{i=0}^{N} \underbrace{x_{k+1} - f(x_k, u_k)}_{\text{defect}}$$

this is called a **merit function**

inequality constraints

- the previous formulation can easily be extended to include **inequality constraints**
- for example, a general constraint

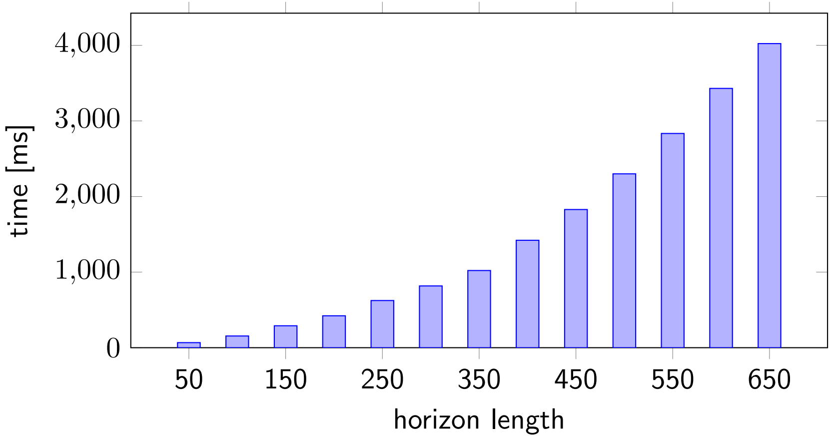
$$g(x_k, u_k) \leq 0$$

could be approximated in a similar fashion

$$g(\bar{x}_k, \bar{u}_k) + \left.\frac{dg}{dx_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k + \left.\frac{dg}{du_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta u_k \leq 0$$

computational complexity

- the expensive step is the **factorization** of a large matrix, which usually scales badly with the number of variables



example

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

n, m = (4, 1)
N = 100
Q = np.eye(n)
R = np.eye(m)
Qter = np.eye(n)
iterations = 10
x_init = np.array([0, 0, 0, 0])
x_goal = np.array([cs.pi, 0, 0, 0])

# initial guess
x = np.zeros((n,N+1))
u = np.zeros((m,N))
x[:, 0] = x_init

# dynamics and its derivatives
f = model.get_pendubot_model()
opt_sym = cs.Opti()
X_ = opt_sym.variable(n)
U_ = opt_sym.variable(m)
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
# optimization problem
opt = cs.Opti('conic')
opt.solver('proxqp')

dX = opt.variable(n,N+1)
dU = opt.variable(m,N)
X = opt.parameter(n,N+1)
U = opt.parameter(m,N)

opt.subject_to( dX[:,0] == np.zeros(n) )
opt.subject_to( X[:,N] + dX[:,N] == x_goal )
for i in range(N):
  opt.subject_to( X[:,i+1] + dX[:,i+1] == f(X[:,i], U[:,i]) + \
                  fx(X[:,i], U[:,i]) @ dX[:,i] + \
                  fu(X[:,i], U[:,i]) @ dU[:,i] )

cost = (X[:,N] + dX[:,N] - x_goal).T @ Qter @ (X[:,N] + dX[:,N] - x_goal)
for i in range(N):
  cost = cost + \
         (X[:,i] + dX[:,i] - x_goal).T @ Q @ (X[:,i] + dX[:,i] - x_goal) + \
         (U[:,i] + dU[:,i]).T @ R @ (U[:,i] + dU[:,i])

opt.minimize(cost)

# SQP iterations
for iter in range(iterations):
  opt.set_value(X, x)
  opt.set_value(U, u)
  sol = opt.solve()
  u = sol.value(U) + sol.value(dU)
  x = sol.value(X) + sol.value(dX)
```

why is SQP good?

- it allows to define **constraints**, which is good because otherwise every objective would be in the cost function (many weights to tune!)
- the number of iterations can be set to a low number to get a suboptimal solution: good for **real-time** operation
- having constraints facilitates writing a **multiple shooting** formulation

why is SQP bad?

- we don't now apriori how it scales with the size of the horizon: it depends on the solver we use for the QP subproblem
- in particular, if we use a dense solver, we usually have to factor a large matrix, which is bad if we have a lot of variables
- if the solver properly exploits the **sparsity** of the problem, it could scale a lot better