EN530.603 Applied Optimal Control
Lecture 9: Numerical Methods for Deterministic Optimal Control

October 21, 2015

Lecturer: Marin Kobilarov

Consider the general setting where we are minimizing

$$J = \phi(x(t_f), t_f) + \int_{t_0}^{t_f} L(x(t), u(t), t)dt, \tag{1}$$

subject to:

$$x(t_0) = x_0, \quad x(t_f) \text{ and } t_f \text{ free} \tag{2}$$

$$\dot{x}(t) = f(x(t), u(t), t) \tag{3}$$

$$c(x(t), u(t), t) \leq 0, \text{ for all } t \in [t_0, t_f] \tag{4}$$

$$\psi(x(t_f), t_f) \leq 0, \tag{5}$$

We will consider three families of numerical methods for solving the optimal control problem:

- *dynamic programming*: i.e. solution to the HJB equations
- *indirect methods* - based on calculus of variations, and Pontryagin's principle
- *direct methods* - based on a finite-dimensional representation, e.g. using a discrete-time formulation or using a parametrized control signal

The distinction between indirect and direct methods lies in whether one first derives continuous optimality conditions and then solves them numerically (indirect) as opposed to first discretizing/parametrizing the problem and then solving it numerically (direct).

1 HJB Solutions

Solving the HJB is equivalent to finding a value function $V(x, t)$ defined over the whole domain $\mathcal{X} \times [t_0, t_f]$ where $x \in \mathcal{X}$ and $t \in [t_0, t_f]$. The control law is then found through the optimization:

$$u^*(x(t), t) = \min_u H(x(t), u, \partial_x V(x(t), t), t)$$

and constitutes a global solution to the optimal control problem from any starting pair $(x(t), t)$. In general, it is extremely difficult to compute V for any nonlinear system. Various approximation methods are applicable.

1.1 Discrete space-time methods

As we already discussed one possible way to solve the HJB equations is to discretize both space and time, convert the problem into a path-finding problem on a graph. Then we can apply the value function expansion methods (value iteration) starting from the goal set backwards. This works for low-order problems with simple dynamics.

1.2 Value function approximation

Another approach is to employ a finite-dimension parametrization of the value function V using e.g. basis functions:

$$V(x,t) = \sum_{i=1}^{L} w_i \varphi_i(x,t),$$

where each $\varphi_i(x,t)$ could be a polynomial (linear, quadratic, etc..,), Gaussian, etc... function. A basis function is chosen so that its partials are simple to compute so that

$$\partial_t V(x,t) = \sum_{i=1}^{L} w_i \partial_t \varphi_i(x,t),$$

$$\nabla_x V(x,t) = \sum_{i=1}^{L} w_i \nabla_x \varphi_i(x,t)$$

have closed-form expressions. Solving the HJB equation now requires the computations of the weights w_i.

This representation is typically employed in *approximate dynamic programming* (ADP) methods often used in reinforcement learning where one might not have access to an underlying model but only to value function evaluations. In such situations it is possible to compute the control u^* purely from the value function (since u depends on $\partial_x V$) and by observing the change in state x. These methods though will not be our focus.

2 Indirect Methods

The starting point is the derivation of the necessary conditions

$$\dot{x} = f(x,u,t) \tag{6}$$

$$\dot{\lambda} = -\nabla_x H(x,u,\lambda,t), \tag{7}$$

$$u^* = \min_u H(x,u,\lambda,t), \tag{8}$$

$$\lambda(t_f) = \nabla_x \phi(x(t_f),t_f) + \nabla_x \psi(x(t_f),t_f)^T \nu, \tag{9}$$

$$\Big(\partial_t \phi + \nu^T \partial_t \psi + H\Big)_{t=t_f} = 0, \tag{10}$$

For now, let's ignore path constraints (4). We will consider three sub-families of indirect methods: indirect shooting, indirect multiple-shooting, and indirect transcription (also known as indirect collocation).

2.1 Indirect Shooting

Shooting methods are based on integrating the EL equations forward from time t_0 to time t_f using a starting guess for the multipliers and then satisfying the boundary and transversality conditions at time t_f by formulating a root-finding problem solved using e.g. Newton-type method.

The optimization variables in this case are

$$\lambda(t_0), \nu, t_f$$

and the equations to be solved are

$$\begin{bmatrix} \psi(x(t_f), t_f) \\ \lambda(t_f) - \left[\nabla_x \phi(x(t_f), t_f) + \nabla_x \psi(x(t_f), t_f)^T \nu\right] \\ \Big(\partial_t \phi + \nu^T \partial_t \psi + H\Big)_{t=t_f} \end{bmatrix} = 0, \tag{11}$$

The pros and cons of the methods can be summarized as follows

Pros.

- low dimension, very efficient solution possible if the EL equations (6) and (7) can be integrated efficiently
- applicable to large-scale systems

Cons.

- *locality*: formulation relies only on necessary conditions, so it is in fact necessary to be able to solve (8) for u^* in terms of x and λ. This is problematic for problems with singularities.
- *sensitivity*: the EL equations are typically highly-nonlinear and small changes in $\lambda(t_0)$ result in huge changes in the final conditions. This is problematic for a gradient-based solver of (11).
- *instability*: when the dynamics (6) is unstable it is extremely difficult to find an initial $\lambda(t_0)$ so that (6) can be integrated numerically
- *path constraints*: difficult to treat since they introduce constraints sub-arcs that must be handled with additional multipliers which must be set up defined a-priori
- *closed-form derivatives:* one actually need to derive the EL equations which means computing analytical derivatives of f, L, ϕ, and ψ that might not be easy to do

A *remedy* to the problem of instability is to actually not to start by blindly guessing a $\lambda(t_0)$ but to start with a reasonable $u(t)$ over $[t_0, t_f]$, then integrate the dynamics (6) forward, and then integrate the adjoint equations (7) backwards using $\lambda(t_f)$ which is known from (9). This is known as the *sweep* method.

2.2 Indirect Multiple-Shooting

Multiple-shooting resolves some of the limitations of single shooting by splitting the time-interval into segments and solving local shooting problems on each segment that are then pieces together by solving addition root-finding problems that *glue* the local solutions together.

First, choose discrete times $[t_0, t_1, t_2, \dots, t_N]$ where $t_N = t_f$ and let $v_i = (x(t_i), \lambda(t_i))$. The idea is to perform a shooting step on each $[t_i, t_{i+1}]$ by integrating the EL equations (6),(7) starting from v_i and obtaining, say $\bar{v}_i$ at time t_{i+1}, for each i. Then we solve the root finding problem The optimization variables in this case are

$$\lambda(t_0), v_1, v_2, \dots, v_N, \nu, t_f$$

and the equations to be solved are

$$\begin{bmatrix} \bar{v}_0 - v_1 \\ \vdots \\ \bar{v}_{N-1} - v_N \\ \psi(x(t_f), t_f) \\ \lambda(t_f) - \left[\nabla_x \phi(x(t_f), t_f) + \nabla_x \psi(x(t_f), t_f)^T \nu\right] \\ \Big(\partial_t \phi + \nu^T \partial_t \psi + H\Big)_{t=t_f} \end{bmatrix} = 0, \tag{12}$$

The pros and cons of the methods can be summarized as follows

Pros.

- *robustness*: issues related to stability are alleviated

Cons.

- *dimensionality*: the problem dimension is increased and becomes more computationally expensive, but one can exploit *sparsity* is the root-finding problem
- *path constraints:* it is still necessary to define constrained-unconstrained arcs in advance

2.3 Indirect Transcription

Similarly to multiple shooting, we choose discrete times $[t_0, t_1, t_2, \ldots, t_N]$ where $t_N = t_f$ and let $v_i = (x(t_i), \lambda(t_i))$, although typically N would be larger here. The idea is to perform a single integration step on each $[t_i, t_{i+1}]$ of the EL equations (6),(7). Denote the combined EL equations by

$$F(v, t) = \begin{bmatrix} f(v, t) \\ -\nabla_x H(v, t) \end{bmatrix},$$

where we assume that $u = u(v, t)$. Then on each segment we can enforce, e.g.

$$v_{i+1} - v_i - h_i F(v_i, t_i) = 0,$$

where $h_i = t_{i+1} - t_i$. This is the simplest Euler-type discretization. A better choice is to us e.g. trapezoidal

$$v_{i+1} - v_i - h_i \frac{F(v_i, t_i,) + F(v_{i+1}, t_{i+1})}{2} = 0,$$

or midpoint

$$v_{i+1} - v_i - h_i F\left(\frac{v_i + v_{i+1}}{2}, \frac{t_i + t_{i+1}}{2}\right) = 0,$$

bot of which are second-order accurate and implicit in v_{i+1}. Higher-order conditions are also possible. The collection of these conditions for all $i = 0, \ldots, N-1$ are solved along with transversality conditions:

$$\begin{bmatrix} S(v_0, v_1) \\ \vdots \\ S(v_{N-1}, v_N) \\ \psi(x(t_f), t_f) \\ \lambda(t_f) - \left[\nabla_x \phi(x(t_f), t_f) + \nabla_x \psi(x(t_f), t_f)^T \nu\right] \\ \left(\partial_t \phi + \nu^T \partial_t \psi + H\right)_{t=t_f} \end{bmatrix} = 0, \tag{13}$$

where $S(v_0, v_1)$ is one of the finite-different schemes such as trapezoidal and midpoint. Matlab's function `bvp2` can be used for solving this class of problems, it uses a fourth-order accurate discretization.

The pros and cons of the methods can be summarized as follows

Pros.

- *accuracy*: solution can be very accurate although no internal integration is performed
- *efficiency*: the problem remains sparse so it can be solved efficiently as long as sparsity is exploited

Cons.

- *closed-form adjoint equations*: still need adjoint equations (i.e. and all derivatives) in closed-form
- *path constraints:* it is still necessary to define constrained-unconstrained arcs in advance

3 Direct Methods

Direct methods are solutions to a finite-dimensional representation of the continuous optimal control problem. This is accomplished either by discretizing time and solving a discrete-time optimal control problems, or by parametrizing the controls using a finite set of parameters such as a polynomial (technically a discrete sequence of controls is also a type of parametrization). Since this becomes a finite-dimensional nonlinear optimization problem the resulting techniques can be regarded as *nonlinear programming* (NLP) methods.

3.1 Direct Shooting

Direct shooting is based on parametrizing the control signal $u(t)$ using a finite number of parameters $p \in \mathbb{R}^c$. For instance, in simple situations (e.g. a one-dimensional double integrator with minimum acceleration cost function), is might suffice to have a control of the form

$$u(t) = p_1 + p_2 t$$

in which case we only need two parameters. More generally, we can employ

$$u(t) = \sum_{k=1}^{M} p_k B_k(t),$$

where M is finite and $B_k(t)$ are a set of basis functions, such as B-splines. The unknowns in the problem are then

$$\text{NLP variables}=[p, t_f]$$

The state $x(t)$ is then obtained using forward integration of the dynamics (6). The NLP problem then becomes

$$J = \phi(x(t_f), t_f) + \int_{t_0}^{t_f} L(x(t), u(t), t), \tag{14}$$

subject to:

$$\psi(x(t_f), t_f) \leq 0, \tag{15}$$

Note that there is no systematic way to incorporate path constraints in the formulation.

The pros and cons of the methods (very similar to indirect shooting) can be summarized as follows

Pros.

- *low dimension*: often difficult large-scale problems can be formulated using relatively low number of parameters p which might yield efficient solution if the EL equations (6) and (7) can be integrated efficiently
- *simplicity*: does not rely on adjoint equations and is very straightforward to implement

Cons.

- *sensitivity*: small changes early in the trajectory result in large deviations at later times and cause problems for gradient based methods which try to stabilize the resulting boundary conditions
- *instability*: when the dynamics (6) is unstable the solution of integrating (6) could also vary substantially also due to numerical round-off errors
- *path constraints*: difficult to treat since they introduce constraints sub-arcs that must be handled with additional multipliers which must be set up defined a-priori
- *loss of sparsity*: the resulting nonlinear equations to be solved are typically complicated and non-sparse so it is not possible to gain efficiency using sparse solvers

3.2 Direct Multiple-Shooting

Direct multiple-shooting is similar to indirect multiple shooting with the only difference that we do not evolve the Lagrangian multipliers.

Choose discrete times $[t_0, t_1, t_2, \ldots, t_N]$ where $t_N = t_f$, as well as discrete trajectory $[x_0, x_1, \ldots, x_N]$ and a discrete set of parametrized control $[u(p_0), u(p_1), \ldots, u(p_N)]$. The we perform a direct shooting step within each interval $[t_i, t_{i+1}]$ by integrating the dynamics (6) starting from x_i and obtaining, say $\bar{x}_i$ at time t_{i+1}, for each i. Then we solve the root finding problem The optimization variables in this case are

$$p_0, p_1, \ldots, p_{N-1}, x_1, x_2, \ldots, x_N, t_f$$

and the equations to be solved are

$$\begin{bmatrix} \bar{x}_0 - x_1 \\ \vdots \\ \bar{x}_{N-1} - x_N \\ \psi(x_N, t_N) \end{bmatrix} = 0, \tag{16}$$

The pros and cons of the methods are similar to indirect shooting

Pros.

- *robustness*: issues related to stability are alleviated
- *efficiency*: sparsity is now introduced through the extra NLP variables since the problem is dense only within segments, and block-sparse overall

Cons.

- *dimensionality*: the problem dimension is increased and becomes more computationally expensive, but one can exploit *sparsity* is the root-finding problem
- *path constraints:* it is still necessary to define constrained-unconstrained arcs in advance (unless a whole interval $[t_i, t_{i+1}]$ can be regarded as either constrained or unconstrained, but this might requires very fine discretization)

3.3 Direct Transcription/Collocation

We start by choosing discrete times $[t_0, t_1, t_2, \ldots, t_N]$ and defining the optimization variables

$$\text{NLP variables} = [u_0, x_1, u_1, \ldots, x_N, u_N, t_f].$$

We replace the nonlinear dynamics (6) with a finite difference approximation, e.g. the trapezoidal rule

$$x_{i+1} - x_i - h_i \frac{f(x_i, u_i, t_i) + f(x_{i+1}, u_{i+1}, t_{i+1})}{2} = 0,$$

or midpoint

$$x_{i+1} - x_i - h_i f\left(\frac{x_i + x_{i+1}}{2}, \frac{u_i + u_{i+1}}{2}, \frac{t_i + t_{i+1}}{2}\right) = 0,$$

or another higher-order method.

Similarly, the cost function is approximated, e.g. by

$$\int_{t_i}^{t_{i+1}} L(x, u, t)dt \approx h_i \frac{L(x_i, u_i, t_i) + L(x_{i+1}, u_{i+1}, t_{i+1})}{2},$$

or

$$\int_{t_i}^{t_{i+1}} L(x, u, t)dt \approx h_i L\left(\frac{x_i + x_{i+1}}{2}, \frac{u_i + u_{i+1}}{2}, \frac{t_i + t_{i+1}}{2}\right),$$

or another higher-order method, respectively. We can then solve the resulting NLP defined by

$$\phi(x_N, t_N) + \sum_{i=0}^{N-1} L_i(x_i, x_{i+1}, u_i, u_{i+1}), \tag{17}$$

subject to:

$$S_i(x_i, x_{i+1}, u_i, u_{i+1}) = 0 \tag{18}$$

$$c(x_i, u_i, t_i) \leq 0, \text{ for all } i = 0, \ldots, N \tag{19}$$

$$\psi(x_N, t_N) \leq 0, \tag{20}$$

where S_i and L_i correspond to the discrete approximations of the dynamics and cost function, e.g. using the midpoint, trapezoidal, or another higher-order scheme.

The solution can generally be computed using sequential quadratic programming (SQP) methods with active set techniques for handling path constraints, or using interior-point (IP) methods. Problem sparsity should be exploited for efficiency.

The pros and cons of the methods can be summarized as follows

Pros.

- *efficiency*: the problem is complex, large-scale but still sparse
- *simplicity*: no need to derive adjoint equations in terms of λ, can use x directly (although technically SQP internally uses multipliers)
- *stability*: unstable dynamics remedied since all x are simultaneously varied
- *path constraints*: easier to handle using existing NLP methods (SQP, IP)

Cons.

- *adaptation*: time grid must be chosen in advance and often difficult to adapt (e.g. where higher accuracy is required) during optimization, since this changes the problem dimension, i.e. more points can't be easily added
- *careful implementation*: to obtain efficiency one still needs derivatives and sparsity structure which must be carefully specified. Recent software packages are starting to provide automated ways to deal with this though.

3.4 Other types of discretization

3.4.1 Constant control along each segment

Often, we can regard the controls u_i as being constant along the segment b/n states x_i and x_{i+1}. In this case the discretization of the dynamics is expressed as

$$S_i(x_i, x_{i+1}, u_i) = 0,$$

since the trapezoidal or midpoint rule will simply use u_i instead of u_{i+1}. In addition, if we solve this equation for x_{i+1} then we the dynamics can be expressed in an explicit form as

$$x_{i+1} = f_i(x_i, u_i)$$

3.4.2 Augmented state using current control

Another strategy is to convert the problem into an equivalent problem operating in an *augmented* state space constructed as the product of the original state space and the control space. More specifically, if we denote $y_i = (x_i, u_i)$ then the dynamics above becomes

$$S_i(y_i, y_{i+1}) = 0,$$

and if one can solve for x_{i+1} then one can formulate it in explicit form by relabeling the controls $v_i \triangleq u_{i+1}$ (for the purpose of preserving meaningiful indexing) as

$$y_{i+1} = \bar{f}_i(y_i, v_i),$$

where $\bar{f}_i(y, v) \triangleq (f_i(x_i), v)$.

4 Direct sweep methods

Sweep methods could be very effective in cases when there are no path constraints. Assume that we have discretized time and approximated the problem by minimizing

$$\begin{gathered} J_0 = L_N(x_N) + \sum_{i=0}^{N-1} L_i\left(x_i, u_i\right), \\ \text{subject to } \ x_{i+1} = f_i(x_i, u_i), \ \text{with given } x(t_0), t_0, t_N \end{gathered} \tag{21}$$

While we do not define general path constraints, control bounds of the form $u \in \mathcal{U}$ where $\mathcal{U}$ is a convex set are applicable.

The standard *pure stage-wise Newton method* [?] sequentially optimizes a Hamiltonian

$$H_i(x, u, \lambda) = L_i(x, u) + \lambda^T f_i(x, u)$$

and is effectively equivalent to the SQP approach applied to this limited class of unconstrained problems. At the core of stage-wise methods lies the standard backward-forward procedure which starts with a given initial trajectory, updates the multipliers λ_i in a backward pass and then adjusts the controls u_i in a forward pass. Backward-forward sweep methods are still considered among the most efficient solutions to optimal control problems especially in the absence of complicated or highly nonlinear inequality state constraints. For arbitrary state constraints it is often easier to use a numerical programming package [?] that might offer special constraint handling functionality such as active-set or interior-point procedures.

Closely related to stage-wise Newton, *differential dynamic programming* (DDP) sequentially optimizes a local model of the Hamilton-Jacobi-Bellman *value function* [?] defined by

$$V_i(x) = \min_{u_{i:N-1}} J_i(x, u_{i:N-1})$$

where $x_i \equiv x$ and $x_{i+1} = f(x_i, u_i)$. It can also be defined recursively according to

$$V_i(x) = \min_u Q_i(x, u) = \min_u \{L_i(x, u) + V_{i+1}\left(f_i(t_i, x, u)\right)\}, \tag{22}$$

where Q is called the unoptimized value function, or Q-function. The difference between pure-SN and DDP lie in second-order terms of the dynamics f_i and in the fact that DDP employs the true nonlinear dynamics in the forward pass [**?**, **?**]. Nevertheless, most implementation of SN are not "pure", i.e. they are easily modified to use the nonlinear dynamics so DDP and SN become more similar. A recent interesting application of DDP to robotics can be found in [**?**].

In DDP one computes the controls u_i to minimize a second-order approximation of Q_i given by

$$Q_i(x_i+\delta x_i, u_i+\delta u_i) \approx Q_i(x_i,u_i) + \frac{1}{2}\begin{bmatrix} 1 \\ \delta x_i \\ \delta u_i \end{bmatrix} \begin{bmatrix} 0 & \nabla_x Q_i^T & \nabla_u Q_i^T \\ \nabla_x Q_i & \nabla_x^2 Q_i & \nabla_{xu} Q_i \\ \nabla_u Q_i & \nabla_{ux} Q_i & \nabla_u^2 Q_i \end{bmatrix} \begin{bmatrix} 1 \\ \delta x_i \\ \delta u_i \end{bmatrix}.$$

By the principle optimality, δu_i is chosen to minimize this approximation which results in the condition

$$\delta u_i^* = K_i \cdot \delta x_i + \alpha_i k_i, \tag{23}$$

where

$$K_i = -\nabla_u^2 Q_i^{-1} \nabla_{ux} Q_i, \quad k_i = -\nabla_u^2 Q_i^{-1} \nabla_u Q_i,$$

and where $\alpha_i > 0$ is a chosen step-size. In addition, there is a relationship between the variations δx_i and δu_i due to the dynamics which is expressed as

$$\delta x_{i+1} = A_i \delta x_i + B_i \delta u_i, \tag{24}$$

where $A_i \triangleq \nabla_x f_i(x_i,u_i)$ and $B_i \triangleq \nabla_u f_i(x_i,u_i)$.

During the backward sweep the terms K_i and k_i are found recursively. During the forward sweep the dynamics is integrated forward with the updated controls. The Backward and Forward sweeps are formulated according to:

```
DDP-Backward
$V_x = \nabla L_N$
$V_{xx} = \nabla_x^2 L_N$
For $k = N-1 \to 0$
    $Q_x = \nabla_x L_i + A_i^T V_x,$
    $Q_u = \nabla_u L + B^T V_x$
    $Q_{xx} = \nabla_x^2 L_i + A_i^T (V_{xx}) A_i$
    $Q_{uu} = \nabla_u^2 L_i + B_i^T (V_{xx}) B_i$
    $Q_{ux} = \nabla_u \nabla_x L_i + B_i^T (V_{xx}) A_i$
    $k_i = -Q_{uu}^{-1} Q_u, \quad K_i = -Q_{uu}^{-1} Q_{ux}$
    $V_x = Q_x + K_i^T Q_u$
    $V_{xx} = Q_{xx} + D^T Q_{ux}$
```

```
DDP-Forward
$\delta x_0 = 0, \qquad V_0' = 0$
For $k = 0 \to N-1$
    $u_i' = u_i + \alpha k_i + K_i \delta x_i$
    $x_{i+1}' = f_i(x_i', u_i')$
    $\delta x_{i+1} = x_{i+1}' - x_{i+1}$
    $V_0' = V_0' + L_i(x_i', u_i')$
$V_0' = V_0' + L_N(x_N')$
```

The step-size α is chosen using e.g. Armijo rule to ensure that the resulting controls $u_{0:N-1}'$ yield a sufficient decrease in the optimal cost $V_0' - V_0$, where $V_0$is the current total cost and V_0' is the new cost at the end of the forward sweep.

The complete algorithm consists of iteratively sweeping the trajectory backward and forward until convergence:

```
Optimize($x_{0:N}, u_{0:N-1}$)
Iterate until convergence
    DDP-Backward
    DDP-Forward
```

Note that a complete implementation of DDP includes regularization steps when computing the iverse of $\nabla_u^2 Q$, similarly to a regularized Newton or a Levenberg-Marquardt algorithm. The Matlab/C++ implementation that you will be given already has this functionality.