Optimization of Mechatronic Systems - Case Studies

Contents

1 Introduction **1**

2 Fitting problems **1**

3 General Optimal Control Problems **3**
3.1 Simple trajectory generation: exploring solution methods 4
3.1.1 Discretization of the system dynamics 4
3.1.2 Single-shooting 6
3.1.3 Multiple-shooting 6
3.1.4 Spline parameterization 7
3.2 More advanced optimal control: validating solution methods 9
3.2.1 Collision avoidance 9
3.2.2 Pendulum 9

4 Trajectory generation using invariant representations **10**

1 Introduction

Optimization problems commonly arise in the daily life of an engineer. During the following case studies we will deal with a few examples, formulate them as optimization problems and solve them. Among these examples are fitting problems, trajectory generation problems and robot learning by demonstration. In general, the followed workflow always consists of the same steps: formulate the optimization problem, pass it to a solver and interpret the solution. During the seminars we will handle each of these steps, and in particular focus on the different ways to formulate a problem.

2 Fitting problems

To provide a clear introduction to the formulation and solution of optimization problems, we will first focus on a very important application of optimization theory: fitting problems. These problems show up in various forms: trendline finding, interpolation of data, marker detection, shape recognition,... We will start with a simple shape-finding problem.

Consider two sets of data points:

$$\text{data} = \begin{bmatrix} 1 & 2-\frac{\sqrt{2}}{2} & 2 & 3 & 2 \\ 0 & \frac{\sqrt{2}}{2} & 1 & 0 & -1 \end{bmatrix}; \text{data\_perturbed} = \begin{bmatrix} 1 & 2-\frac{\sqrt{2}}{2} & 2 & 2.5 & 2 \\ 0 & \frac{\sqrt{2}}{2} & 1 & 0 & -1 \end{bmatrix}. \tag{1}$$

Our aim is to find the circle with radius 1 and unknown centerpoint p that *best* matches the unperturbed data.

Exercise A.1 *Think about a way to formulate this task as an unconstrained optimization problem. Suggest an objective. Think in terms of the gaps that are constructed in Figure 1*
For these exercises, we will work with CasADi's Opti environment. You find an example of the syntax below:

```
opti = casadi.Opti();

x = opti.variable(1,1);
```

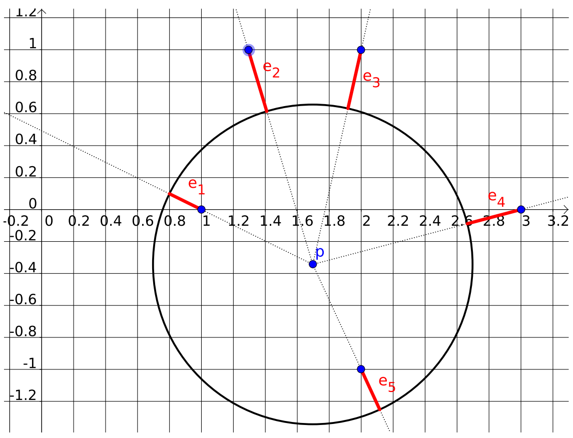


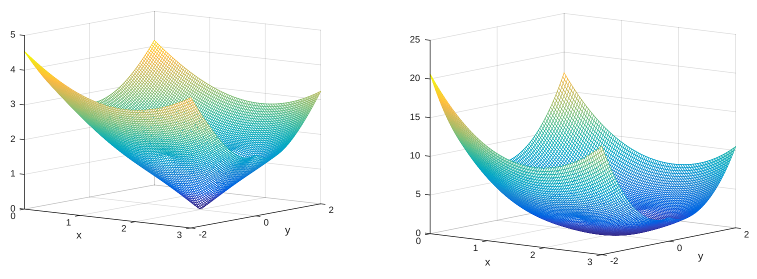
Figure 1: Geometric construction of gaps e_i for a circle fitting problem.

```
y = opti.variable(1,1);

opti.minimize((1-x)^2+100*(y-x^2)^2);
opti.subject_to(x^2+y^2<=1)
opti.subject_to(x+y>=0)

opti.solver('ipopt');
sol = opti.solve();

sol.value(x)
sol.value(y)
```

Exercise A.2 *Obtain CasADi and the users guide (inside the example pack; only the chapter about 'Opti' is relevant for you) from* ***http://install33.casadi.org****. Type the above example in Matlab and verify that it runs without errors.*

Exercise A.3 *Using the Opti environment, formulate and solve your circle fit problem.*

Exercise A.4 *In Figure 2, you will find two subtly different objectives. Which one will give you the most optimal fit? Which one will be easiest to solve by a gradient-based optimizer like Ipopt? Try both in Matlab and compare the results.*



(a) $\underset{p}{\text{minimize}} \quad ||e(p)||_2$ (b) $\underset{p}{\text{minimize}} \quad ||e(p)||_2^2$

Figur Two objectives for a circle fit.

Next, we will consider an $L1$-norm. Since it is not smooth, this objective can't be handled well by a gradient-based optimizer. Therefore, we propose a reformulation using a slack variable:

$$\underset{p}{\text{minimize}} \quad ||e(p)||_1 \iff \begin{array}{ll} \underset{p,s_i}{\text{minimize}} & \sum s_i \\ \text{subject to} & -s_i \leq e_i(p) \leq s_i \end{array} \tag{2}$$

Exercise A.5 *Implement the $L1$-reformulation for the circle fit problem. Hint: you may wish to use some initial guess for p as follows:* `opti.set_initial(p,[1.5;0.5])`*. Compare the quality of the $L1$-fitting formulation with that of the previous $L2$.*

Exercise A.6 *Repeat the numerical fitting using $L1$ and $L2$ norms, using the perturbed data. Again, compare the two formulations qualitatively.*

For the second example we consider an extension to our original problem, towards the domain of image processing. Previously, our data consisted of a set of measurements of the contours of a circle. Now, suppose that we make a movie of the movement of a certain object (e.g. a marker attached to a robot arm), and that we want to obtain the position of this object at each point in time. To simplify the situation, we created an animation of a red circle that is moving through a frame with a black background. In a first step, we extract all frames of the movie and save them as .png-files.

Exercise A.7 *One intuitive solution could be to search for the 'most red pixel'. When there is no noise present, this will just give you a random pixel of the circle, while we are searching for the center. When there is noise present, this pixel may not even lie inside the circle. Therefore, we try to implement a more intelligent method. Suppose that we know that the radius of the circle is 22 pixels and that we use* ***Image0001.png****. Find the circle center as the combination of the row and column with the largest amount of red pixels. Use the support files* ***tracking.m****,* ***draw_center.m*** *and* ***draw_circle.m****, and implement your method in* ***tracking_method_2.m****. Does this work well?*

Exercise A.8 *In the previous two methods we didn't use explicit optimization anymore, and the results were not fully as desired. Therefore, for the third method, find the circle position as the mass center of the non-black pixels:*

$$\underset{center}{\text{argmin}} \int \|data - center\|^2 d(data). \tag{3}$$

For this case, the integral over the complete image is discretized as a sum over the data, and the center is found as the position that gives you the least square error. Again, use the Opti environment. Suppose that you know that the circle never touches the boundary of the image frame. Translate this knowledge into constraints that make the work of the solver easier.

Exercise A.9 *Use the three methods to loop over all frames of the movie, in order to obtain a set of position measurements. Can you think of a way to make the work of the solver easier?*

Exercise A.10 *Re-run all your methods for the case in which there was noise added to the frames. Do they all work as well as before?*

3 General Optimal Control Problems

Computing an optimal motion trajectory always comes down to solving an Optimal Control Problem (OCP). This means that we look for the sequence of inputs $\mathbf{u}(t) \in U$ that bring the system optimally to its desired goal state, along a state trajectory $\mathbf{x}(t) \in X$, while fulfilling the imposed constraints at all times. In addition, the obtained input and state trajectories minimize a certain objective.

In continuous time, a general OCP is given by:

$$
\begin{aligned}
\underset{\mathbf{x}(t),\mathbf{u}(t)}{\text{minimize}} \quad & \int_0^T l(\mathbf{x}(t),\mathbf{u}(t))dt + E(\mathbf{x}(T)) \\
\text{subject to} \quad & \text{system dynamics}, \quad t \in [0,T] \\
& \mathbf{x}(0) = \mathbf{x}_0, \quad \mathbf{x}(T) = \mathbf{x}_T, \\
& \mathbf{x}(t) \in X, \\
& \mathbf{u}(t) \in U,
\end{aligned} \tag{4}
$$

in which T is the end time, $l(\cdot)$ is the stage cost and $E(\cdot)$ is the terminal cost. The system dynamics typically take the form of an explicit first-order ode:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t,\mathbf{x}(t),\mathbf{u}(t)), \tag{5}$$

or a first-order implicit form:

$$\mathbf{f}(t,\dot{\mathbf{x}}(t),\mathbf{x}(t),\mathbf{u}(t)) = 0. \tag{6}$$

In mechanics applications, a second-order form is also commonly used:

$$\mathbf{f}(t,\ddot{\mathbf{x}}(t),\dot{\mathbf{x}}(t),\mathbf{x}(t),\mathbf{u}(t)) = 0. \tag{7}$$

Note that there is an important difference between a trajectory and a path. A path consists of a sequence of waypoints (positions) that the vehicle needs to follow, and is only defined in space. A trajectory on the other hand, also includes timing information: it tells when the vehicle has to be at which position. As a consequence, a trajectory also includes information about the vehicle velocity at each point in time.

3.1 Simple trajectory generation: exploring solution methods

To start with a very simple example, we compute the motion trajectory $\mathbf{q}(t)$ for a car that has to move as fast as possible from its initial position A to its goal position B. The vehicle starts from and ends in standstill, and is subject to velocity constraints ($v \in [v_{min}, v_{max}]$) and force constraints ($F \in [F_{min}, F_{max}]$), neglecting friction. We will model the vehicle as a point mass which is steered by force inputs.

Exercise B.1 *Formulate the optimization problem on paper, i.e.: write down the dynamic model of the car, together with all required constraints. Think about other, non-necessary constraints that can simplify the work of the solver.*

Since this OCP contains constraints that need to hold over the complete time horizon, i.e.: $t \in [0,T]$, there are an infinite amount of constraints. In addition, the trajectory itself also consists of an infinite amount of points. To handle these problems, and solve the example, we will consider several solution methods: single shooting, multiple shooting and a spline-parameterization. The first two approaches discretize the state and control input, the last approach proposes a parameterization of the trajectories in the form of a spline function.

3.1.1 Discretization of the system dynamics

To obtain a representation of the trajectories with a finite amount of variables, we use a certain parameterization of the state and input trajectories. When using shooting methods, the input is only changed at certain points in time, between those points the input is kept constant. This approach gives us a piecewise constant input trajectory. The resulting (approximate) evolution of the states is obtained by using a numerical integration scheme. Two commonly used schemes are the forward Euler integrator and the Runge Kutta integration schemes.

Euler integration is efficient but not very reliable for non-linear dynamics. This integrator calculates the value of the state at the next time sample $(k+1)$, based on the current time sample (k) according to:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h \cdot \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k), \tag{8}$$

where h is the time step. On the other hand, Runge Kutta integration is much more reliable but also computationally more expensive. The Runge Kutta integration scheme of order 4 is given by:

$$\begin{aligned}
k_1 &= \mathbf{f}(\mathbf{x}_k,\ \mathbf{u}_k),\\
k_2 &= \mathbf{f}(\mathbf{x}_k + \frac{h}{2} \cdot k_1,\ \mathbf{u}_k),\\
k_3 &= \mathbf{f}(\mathbf{x}_k + \frac{h}{2} \cdot k_2,\ \mathbf{u}_k),\\
k_4 &= \mathbf{f}(\mathbf{x}_k + h \cdot k_3,\ \mathbf{u}_k),\\
\mathbf{x}_{k+1} &= h \cdot \frac{1}{6}(k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4).
\end{aligned} \tag{9}$$

Exercise B.2 *In order to investigate the performance of both integration schemes, simulate the following second order system: $\ddot{x} + x = 0$ starting from $x(0) = 1, \dot{x} = 0$ for 10 seconds and using 50 integration steps. Compare the state trajectories obtained by both integration methods with those of the exact solution of the differential equation. Figure 3 plots the error for a range of simulation steps. Explain what you see.*

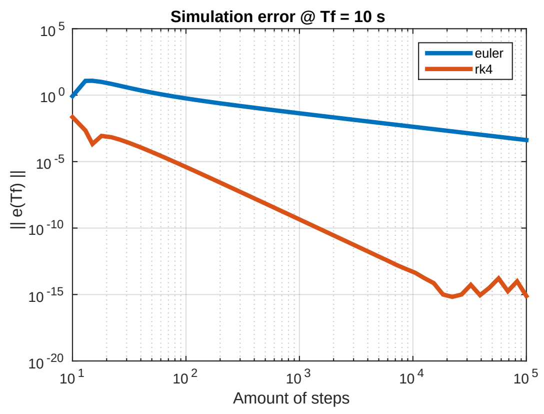


Figure 3: Error appearing in a Runge-Kutta and Euler simulation.

Using an integration method, we can transform the OCP from continuous time, to discrete time, leading to:

$$\begin{aligned}
\underset{x_1,u_1,\ldots,u_N,x_{N+1}}{\text{minimize}} \quad & \sum_{k=1}^{N} L(\mathbf{x}_k, \mathbf{u}_k) + E(\mathbf{x}_{N+1})\\
\text{subject to} \quad & \mathbf{x}_{k+1} - \phi(\mathbf{x}_k, \mathbf{u}_k) = 0,\\
& \mathbf{x}_1 = \mathbf{x}_0, \quad \mathbf{x}_{N+1} = \mathbf{x}_T,\\
& \mathbf{x}_j \in X\\
& \mathbf{u}_k \in U\\
& k = 1 \ldots N, \quad j = 1 \ldots N+1
\end{aligned} \tag{10}$$

3.1.2 Single-shooting

Direct methods generate a feasible search direction in each step, in which the objective function value can be improved. A popular method in the direct method family, is the single-shooting approach (also called sequential approach). In this method, the state and control input are discretized over a finite grid of N samples. The initial state ($\mathbf{x}_1$) and the input at every sample (U) are used to compute all other states, by integrating the vehicle dynamics, using numerical integration techniques (see Section 3.1.1). The advantage of the single-shooting method is that the number of variables is lower, but the relation between variables is more complicated than in the multiple-shooting approach (due to the substitution of the dynamics). When using single-shooting, the OCP from (12) is transformed into:

$$\begin{aligned} \underset{\mathbf{x}_1,\bar{\mathbf{u}}}{\text{minimize}} \quad & \sum_{k=1}^{N} L(\mathbf{x}_k(\mathbf{x}_1,\bar{\mathbf{u}}),\mathbf{u}_k) + E(\mathbf{x}_{N+1}(\mathbf{x}_1,\bar{\mathbf{u}})) \\ \text{subject to} \quad & \mathbf{x}_1 = \mathbf{x_0}, \quad \mathbf{x}_{N+1}(\mathbf{x}_1,\bar{\mathbf{u}}) = \mathbf{x_T}, \\ & \mathbf{x}_j(\mathbf{x}_1,\bar{\mathbf{u}}) \in X \\ & \mathbf{u}_k \in U, \\ & k = 1\dots N, \quad j = 1 \dots N+1 \end{aligned} \tag{11}$$

in which $L(\mathbf{x}_k(\mathbf{x}_1,\bar{\mathbf{u}}),\mathbf{u}_k)$ represents the substitution of the dynamics, by applying the differential equations, the initial state ($\mathbf{x}_1$) and the inputs at every point in time ($\mathbf{u}_k$), and $\bar{\mathbf{u}}$ is the vector of all applied inputs.

Exercise B.3 *Apply this technique to our optimization problem in which a car has to move as fast as possible from A to B. Write the car dynamics into a first-order ode. In addition, formulate the necessary input and state constraints. Compare simple Euler integration with a Runge-Kutta integration scheme of order 4. As numerical values use:* $x_0 = [0,0], x_T = [10,10]$, $v \in [-10,10]\frac{m}{s}$, $F \in [-2500,2500]N, m = 500kg, N = 100$. *Use CasADi's Opti environment to simplify the modeling.*

3.1.3 Multiple-shooting

Another direct method is the multiple-shooting approach (also called simultaneous approach). Crucially for multiple-shooting is that the states at different points in the time-grid are kept as decision variables. The system dynamics are imposed using numerical integration techniques, represented as a integrator function ϕ, yielding the following optimization problem:

$$\begin{aligned} \underset{x_1,u_1,\dots,u_N,x_{N+1}}{\text{minimize}} \quad & \sum_{k=1}^{N} L(\mathbf{x}_k,\mathbf{u}_k) + E(\mathbf{x}_{N+1}) \\ \text{subject to} \quad & \mathbf{x}_{k+1} - \phi(\mathbf{x}_k,\mathbf{u}_k) = 0, \\ & \mathbf{x}_1 = \mathbf{x}_0, \quad \mathbf{x}_{N+1} = \mathbf{x}_T, \\ & \mathbf{x}_j \in X \\ & \mathbf{u}_k \in U \\ & k = 1\dots N, \quad j = 1 \dots N+1 \end{aligned} \tag{12}$$

Exercise B.4 *Apply this technique to the problem in which a car has to move time-optimally over a straight line. Again use Optistack to simplify the modeling. Compare Euler integration with Runge Kutta integration.*

3.1.4 Spline parameterization

A different method to render the motion planning problem tractable, i.e. find both a representation of the trajectory with a finite amount of variables and obtain a finite amount of constraints, is to propose a B-spline parametrization for $\mathbf{q}$. The function $\mathbf{q}$ is then parametrized in terms of a finite function basis $B^{\mathbf{q}}(t) : \mathbb{R} \to \mathbb{R}^m$:

$$\mathbf{q}(t) = \sum_{i=1}^{N} c_i^{\mathbf{q}} \cdot B_i^{\mathbf{q}}(t),$$

in which m is the dimension of $\mathbf{q}$, $c_i^{\mathbf{q}}$ are the spline coefficients and N is determined by the degree of the basis functions and the amount of knots. The spline $\mathbf{q}(t)$ is a piecewise polynomial function, meaning that inside each knot interval (i.e. the space between two subsequent knots) the curve is described by a polynomial. At the border of the knot intervals (i.e. the knots themselves), the different polynomials are connected to each other, with a continuity equal to the degree of the spline minus one. This means that, for a spline of degree two, in a connection point the polynomials themselves have the same value, and their first derivatives are equal. Figure 4 gives a graphical illustration of a spline. The bottom part shows the basis functions, that are scaled by the spline coefficients and, after summation, form the resulting spline thet is shown in the top part. The knot intervals are represented by the alternating blue and white rectangles.

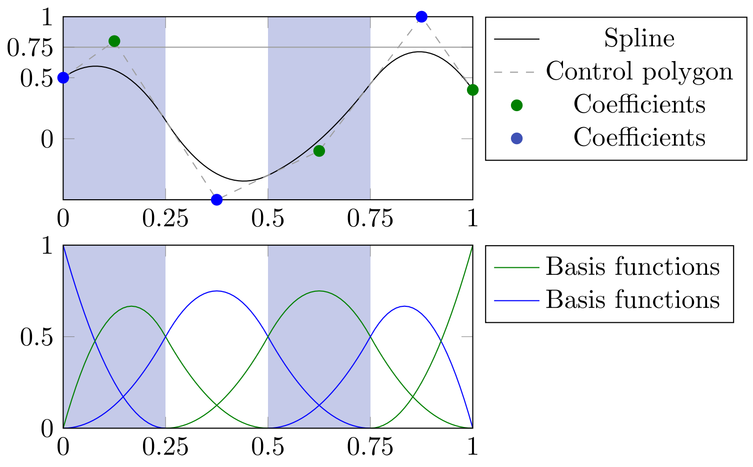


Figure 4: Graphical illustration of a B-spline

When using a spline parameterization for the trajectory, the spline coefficients become the variables of the optimization problem. This gives us a numerically stable way to represent the motion profile $\mathbf{q}(t)$. However, the problem we now face is that the basis functions are a function of time t. We would like to e.g. impose conditions at the end of the curve, i.e. $\mathbf{q}(T)$, which causes problems since T is a variable itself, and therefore unknown when constraints are constructed. Therefore, we transform the time axis by using the substitution $\tau = t/T$, such that $\tau \in [0, 1]$, giving:

$$\mathbf{q}(\tau) = \sum_{i=1}^{N} c_i^{\mathbf{q}} \cdot B_i^{\mathbf{q}}(\tau).$$

Using a spline parameterization leads to various advantages. A first advantage of a spline is that its derivatives can be computed very efficiently:

$$\dot{\mathbf{q}}(t) = \frac{d\mathbf{q}}{d\tau}\frac{d\tau}{dt} = \frac{1}{T} \sum_{i=1}^{N-1} c_i^{\dot{\mathbf{q}}}(c_i^{\mathbf{q}}) \cdot B_i^{\dot{\mathbf{q}}}(\tau),$$

in which there is a linear dependence between the coefficients of the spline derivative and those of the original spline. This gives us a representation of a motion trajectory, and its derivatives using only the coefficients of the original spline as decision variables. Another advantage is that the user gains control over the obtained accuracy of the trajectory: when choosing a higher degree, or a spline with a larger number of knots, the freedom of the trajectory increases and therefore it can more closely resemble the optimal trajectory. Finally, using a spline parameterization allows avoiding so-called time gridding. All previous methods dealt with constraints like:

$$v_{min} \leq \dot{x}(t) \leq v_{max}, \quad \forall t \in [0, T]$$

by making a time grid and only imposing the constraints at these specific points in time. As a consequence, there may be constraint violations between grid points. To reduce this effect, a very fine grid is required, leading to a very high number of constraints. When using a B-spline parameterization it can be shown that the spline always lies inside the convex hull of the connection of the different coefficients (i.e. the control polygon). Therefore, when only imposing constraints on the coefficients, constraint satisfaction is guaranteed at all times, and there is no need for time gridding. The downside is that this technique introduces a little conservatism: instead of constraining the spline itself, we constrain the coefficient, but as shown in Figure 4 there is some distance between the coefficient and the spline. E.g. $x(\tau) \leq 0.75, \forall \tau \in [0, 1]$ would be violated when imposing constraints on the coefficients, while the spline itself does fulfill the constraint.

To efficiently formulate the optimization problem with a B-spline parametrization of x you can use the spline toolbox that was developed at KU Leuven, see: `https://somewebsite.be`. Download this toolbox and add it to the Matlab path. This toolbox offers an Opti-like syntax to make spline variables. The basic commands are shown below:

```
meco_binaries('cpp_splines','refactor')
import splines.*  % import spline toolbox

opti = OptiSpline(); % make spline environment

%Spline parametrization
d      = 3  % spline degree
n      = 11  % total number of knots
L      = 1  % spline domain [0,L]

B      = splines.BSplineBasis([0 , L], d, n);  % make B-spline basis
x      = opti.Function(B, [1, 1]);  % movement spline (1 by 1) with unknown coefficients
dx     = x.derivative(1);  % velocity spline
T      = opti.variable(1);  % motion time
% variable in opti environment beware of the dot!

time = linspace(0,10,10);
values = linspace(0,5,10);
x_guess = splines.Function.linear(time, values);  % initial guess function
T_guess = 10;
% opti.Function gives a function with unknown coefficients
% splines.Function is used for a function with known coefficients
...

%Optimize
opti.subject_to(T>=0);% constraints
...
opti.minimize(T);
% make solver
opti.solver('ipopt');
opti.set_initial(T, T_guess); % assign init_guess

sol = opti.solve();

%Postprocess
x = sol.value(x); %convert symbolic spline to a numeric one
t = 0:0.01:1;  % time vector
x_values = x.list_eval(t);  % evaluate with a vector
...
```

Exercise B.5 *To get familiar with the toolbox and with the concept of splines, run the example code below and play around with it. E.g. change the amount of knots, the degree, plot the coefficients,...*

```
degree = 3;
n_knots = 5;

B = splines.BSplineBasis([0,1], n_knots, degree);
coeffs = randn(B.dimension(),1); %generate random coefficient values
s = splines.Function(B, coeffs); %define spline

t = [0:0.01:1]; %time vector
y = s.list_eval(t); %evaluate spline in every t
figure(1)
plot(t,y) %plot spline

b = B.list_eval(t); %evaluate basis functions in every t
b = full(b); %sparse to full matrix

figure(2)
hold all
for i = 1:size(b,2)
    plot(t,b(:,i))  % plot basis functions
end
```

Exercise B.6 *Solve the studied problem for minimal motion time by using the spline toolbox. This time, use the system dynamics in a second-order implicit form, i.e. use the trajectories itself as variables.*

3.2 More advanced optimal control: validating solution methods

In the previous exercises we have solved the very simple problem of computing the time-optimal movement of a car over a straight line, using different solution methods. In the following exercises we will look at more realistic optimal control problems.

3.2.1 Collision avoidance

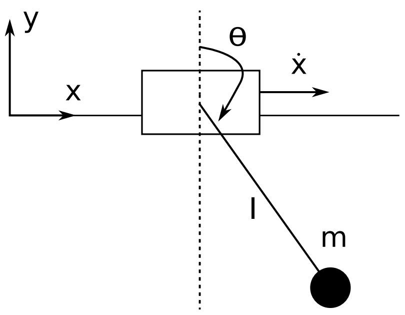
In reality, there will often be obstacles present, making it impossible for the vehicle to follow a straight line from its initial position to its goal position. Each obstacle will lead to extra constraints that are added to the original OCP. In general, collision avoidance constraints express that the shape of the vehicle and that of the obstacle cannot overlap.

Exercise B.7 *Suppose that there is are two obstacles with a circular shape and radius* $1m$ *at position* $[3, 2]$ *and* $[6, 7]$*. Compute a time-optimal, collision-free trajectory for the vehicle by formulating a collision avoidance constraint. Use multiple shooting and compare it with a spline-parameterized trajectory. Play around with the obstacle position and amount of obstacles. Also try to place an obstacle at* $[5, 5]$*, does this problem solve easily for both cases? For multiple shooting, study the effect of* N*.*

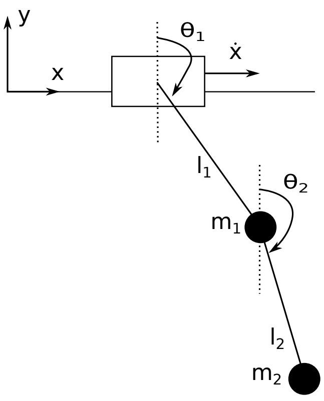
3.2.2 Pendulum

A totally different problem which is actually an OCP is the optimal swing-up of a double pendulum. During the following exercises we will gradually build up the complexity of the problem, by starting with a single pendulum on slider.

Exercise B.8 *Write down the differential equations for a single pendulum attached to a cart which moves over a slider (see Figure 5a), e.g. using the Lagrangian:* $L = E_{kin} - E_{pot}$*. Afterwards, formulate the OCP to swing up the pendulum time-optimally by applying accelerations to the cart. As discretization method, select multiple shooting.*



(a) Single pendulum on a cart



(b) Double pendulum on a cart

Figure 5: Illustration of the considered pendulum set-ups

Exercise B.9 *Consider a double pendulum that is attached to a stationary cart and is controlled by applying an angular acceleration to the base joint. Again formulate and solve the OCP, using multiple shooting.*

Exercise B.10 *Finally, consider a double pendulum that is attached to a moving cart, and is controlled by applying accelerations to the cart. See Figure 5b for an illustration of the set-up. Use the provided descriptions of $\ddot{\theta}_1$ and $\ddot{\theta}_2$ as a function of the states and the inputs (see double_pendulum_ode.m).*

4 Trajectory generation using invariant representations

In various practical applications the trajectory that the autonomous system needs to follow can first be demonstrated by a human, e.g. for a robot that is pouring a drink or building a brick wall. Of course, the robot can't always use the exact same trajectory. To build up a wall numerous similar trajectories are required, with different end positions. This means that the robot has to repeat the demonstrated trajectory, with slight adaptations. There are various approaches to compute an adapted trajectory, based on a demonstrated one. Mostly, it is important that the method returns a trajectory that resembles the original one as good as possible, e.g. in human-robot interaction or in collaborative applications, you want the trajectories to remain interpretable and predictable by the human. We will first explore a very simple and intuitive approach to compute a position trajectory $p(t)$ that is similar to the demonstrated one $p_{\text{ref}}(t)$.

Exercise C.1 *Consider the translational motion $p_{ref}(t) = \begin{bmatrix} t & 0.1t\sin(4\pi t) & 0.1tcos(4\pi t)\end{bmatrix}^T$, for $t = [0, 1]$s, of a point mass. Compute a new trajectory that has a minimum mean square error on the position, and which is constrained to end up at a new target $p(1) = \begin{bmatrix}1 & 1 & 1\end{bmatrix}^T$. Solve this using multiple shooting, with a Runge-Kutta integrator and a timestep of 0.01 s. Use the jerk (third derivative of position) as an input, i.e. suppose that the vehicle dynamics are given by a triple integrator. Add a regularization on the jerk to the objective: $1 \cdot 10^{-7} \sum_k ||u_k||_2^2$.*
Interpret the result, is it desirable or not?

The previous exercise showed that minimizing the difference with a given trajectory in an intuitive way doesn't always give the desired results. In addition, when obtaining a trajectory by measuring the position of a marker, the measurements of course depend on where you placed the marker. Furthermore, the position of the used coordinate system also influences the measurements. Also, it is often very time consuming or even impossible to demonstrate all possible situations to the robot. In order to overcome these problems, we will look at a so-called invariant trajectory description.

This invariant trajectory description is a type of generalization: you want to obtain a general trajectory for a certain task, applicable in various specific cases. Therefore, generalization involves identifying and extracting the parts of the motion that are essential for its execution from the parts that only depend on the context in which the motion is executed. When adapting the motion to new situations, the essential parts must be preserved, while new contextual information must be taken into account. Examples of new contextual information include a new starting location of the motion, a new target location, a different reference frame, a different execution speed, or avoidance of obstacles in the environment.

In this case study we will focus on the so-called Frenet-Serret motion invariants. There are three invariants for translation:

- $i_1(t)$ magnitude of the translational velocity of a reference point on the object in m/s,
- $i_2(t)$ curvature speed in rad/s (deviation of trajectory from a straight line),
- $i_3(t)$ torsion speed in rad/s (deviation of trajectory from the plane),

and three for rotation:

- $i_4(t)$ magnitude of the rotational velocity of the object in rad/s,
- $i_5(t)$ rotational curvature speed in rad/s (deviation from rotation around one (fixed) axis),
- $i_6(t)$ totational torsion speed in rad/s (deviation from rotation around two axes).

These motion invariants offer a complete and minimal representation of the pose trajectory, such that it can be reconstructed given a few parameters such as the initial position and orientation.

The dynamics of translation are given by p (the object position), and R_t (the translational Frenet-Serret frame):

$$\begin{aligned} \dot{p}(t) &= R_{\mathrm{t}}(t)\begin{bmatrix} i_1(t) & 0 & 0 \end{bmatrix}^T, \\ \dot{R}_t(t) &= R_t(t)\mathrm{skew}(\begin{bmatrix} i_3(t) & i_2(t) & 0 \end{bmatrix}^T). \end{aligned} \tag{13}$$

The dynamics of rotation are given by R (the object frame), and R_r (the rotational Frenet-Serret frame)

$$\begin{aligned} \dot{R}(t) &= \mathrm{skew}(R_r(t)\begin{bmatrix} i_4(t) & 0 & 0 \end{bmatrix}^T)R(t), \\ \dot{R}_r(t) &= R_r(t)\mathrm{skew}(\begin{bmatrix} i_6(t) & i_5(t) & 0 \end{bmatrix}^T). \end{aligned} \tag{14}$$

All frames are assumed orthogonal in this description. Skew represents an operator that turns a vector into a skew-symmetric matrix A, meaning that $A^T = -A$.

Exercise C.2 *We will first get acquainted with integrating the Frenet-Serret invariants. Consider a purely translational motion described for* $t = 0 \dots 1\,s$, *by* $i_1(t) = 2, i_2(t) = -10, i_3(t) = -8$. *Plot the corresponding cartesian-space motion for these invariants by integrating the dynamics using Runge-Kutta, with a timestep of* $0.01\,s$. *Verify that you obtain a screw-like motion.*

Exercise C.3 *Find the translational invariants corresponding to the reference motion of exercise C.1, using a multiple-shooting OCP transcription with least-squares objective. Use the same discretization method and step as in the previous exercise. Introduce a regularization term in the objective:*
$1 \cdot 10^{-4} \sum_k || \begin{bmatrix} i_1[k+1] - i_1[k] & i_2[k+1] - i_2[k] & i_3[k+1] - i_3[k] \end{bmatrix} ||_2^2$.

Exercise C.4 *Use the identified invariants to recreate a similar motion in another context: between* $p(0) = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$ *and* $p(1) = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T$. *Again, use a least-squares objective. Verify visually that the motion is similar.*