UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Practical Robotics

Callen Fisher

2021

Aim of course:

1. Learn Lagrange Dynamics
2. Become competent in modelling systems
3. Learn Trajectory optimization methods

Revision History:

1. V1 basic content added, Feb 2021

Disclaimer: Large quantities have been copied from my PhD titled: Trajectory Optimisation Inspired Design for Legged Robotics.

Chapter 1

Dynamics

In robotics, the equations of motion (EoM) are typically generated using Euler-Lagrange dynamics in the form of the manipulator equation:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) = \mathbf{B}\boldsymbol{\tau} + \mathbf{A}\boldsymbol{\lambda} \tag{1.1}$$

Euler-lagrange dynamics is a re-formulation of Newtons 3 laws of motion, and is especially useful in optimization problems. The manipulator equation is the state space representation of the EoM, and is in the correct form for optimization methods.

In the above equation, $\mathbf{q}$ is the generalised coordinates of the robot (such as the position and orientation of each link, using absolute or relative angles) along with the generalised forces and torques represented by $\boldsymbol{\tau}$. The generalised forces and torques typically contain the robots actuators, such as motors, which can apply a torque or force to the robot. $\mathbf{M}$ is the mass matrix, $\mathbf{C}$ is the Coriolis matrix, $\mathbf{G}$ is the gravitational potential matrix, $\mathbf{B}$ maps the applied torques and forces to the generalised coordinates and $\mathbf{A}$ maps the external forces (such as ground reaction forces, λ, broken into horizontal and vertical components) to the generalised coordinates.

There are two approaches used in trajectory optimization, some prefer maximal coordinates, others prefer generalized coordinates. Until proven otherwise, we will use generalized coordinates as they are easier to optimise. Generalized coordinates are the minimum number of coordinates required to define a system. Maximal coordinates are the maximum number of coordinates to define a system.

For a common two link pendulum, generalized coordinates will contain the two angles of the links (either absolute or relative). Maximal coordinates will contain the two angles of the link, as well as the X and Y position of the CoM of each link. If you know the length of each link, and the angles of the links, you can calculate the CoM of each link, therefore for generalized coordinates, you do not require the X and Y position, and only the angles.

In order to generate the EoM, the position of the CoM of each robot link is required (using either relative or absolute coordinates). The potential and kinetic energy of the system can then be defined, which is used to generate the manipulator equation.

1.1 Absolute VS Relative Coordinates

Each system has its pros and cons.

Absolute Coordinates:

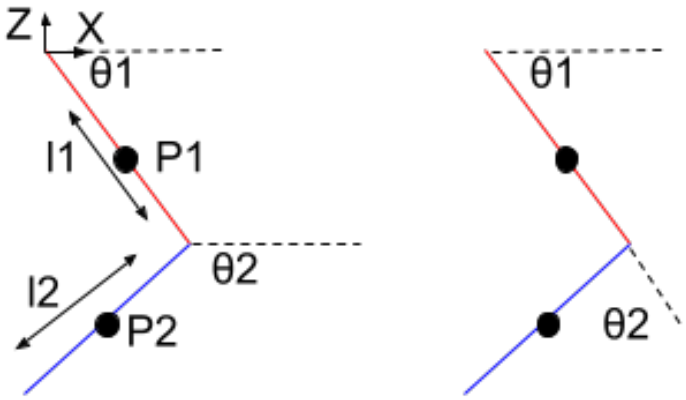

Figure 1.1: Absolute angles are on the left (with respect to the horizontal) with relative angles on the right.

- Simpler dynamics (optimiser potentially solves faster)
- More complex constraints (for example you cannot simply constrain the motion of a torque actuator or link using bounds, you will have to add the constraint to the relative angle). See more information below.

Relative Coordinates:

- More complex dynamics
- Simpler constraints.
- more intuitive for more complex problems

In terms of constraints, a common constraint in robotics, is to limit the motion of a link. For example, if you have a robot arm or leg, it will have a range of motion. In order to enforce this in trajectory optimisation, you need to use bounds or constraints. These act on the relative angle, therefore if you are using relative angles, you can simply use variable bounds (θ_{motor} variable can only vary between its range of motion). However, if you are using absolute angles, you will need to first calculate the relative angle, and apply a constraint onto this.

This does not seem too complex, however if you are implementing a motor model or controller, you need to ensure you are using the correct position and velocity (the position and velocity of the motor and not the absolute position and velocity of the links).

The dynamics of the system depend on how you calculate the position of the CoM of each link. If you are using absolute angles, the position is easier to define, however if you are using relative angles, the position is more complex as seen below:

$$\begin{aligned} P2_{absolute} &= \begin{bmatrix} cos(\theta_1)l_1 + cos(\theta_2)l_2/2 \\ 0 \\ -sin(\theta_1)l_1 - sin(\theta_2)l_2/2 \end{bmatrix} \\ P2_{relative} &= \begin{bmatrix} cos(\theta_1)l_1 + cos(\theta_1+\theta_2)l_2/2 \\ 0 \\ -sin(\theta_1)l_1 - sin(\theta_1+\theta_2)l_2/2 \end{bmatrix} \end{aligned} \tag{1.2}$$

(note the middle coordinate, y, is not required for the 2 dimensional case and can be removed). This can be simplified using rotational matrices as follows:

$$\begin{aligned} P2_{absolute} &= R(\theta_1)[l_1,0,0]' + R(\theta_2)[l_2,0,0]'/2 \\ P2_{relative} &= R(\theta_1)[l_1,0,0]' + R(\theta_1+\theta_2)[l_2,0,0]'/2 \end{aligned} \tag{1.3}$$

where $R()$ is the rotation matrix:

$$R = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{1.4}$$

As soon as you move to 3 dimensions, with more degrees of freedom (roll, pitch and yaw), your rotation matrix becomes more complex. You need to find the position of each CoM for each link used.

add quaternions and rotation matrix with rot order

1.2 Calculating the Energy of the System

Once the positions of each link have been determined, you can calculate the energy of the system. First the velocity of each CoM of each link is required. This is easily calculated as follows:

$$\dot{\mathbf{P}} = jacobian(\mathbf{P},\mathbf{q})\dot{\mathbf{q}} \tag{1.5}$$

The potential energy of the system is the sum of the potential energy of each link:

$$V = \sum m_i \times \mathbf{g} \times \mathbf{P}_i \tag{1.6}$$

where m_i is the mass of the i^{th} link, $\mathbf{g}$ is the gravitational vector and $\mathbf{P}_i$ is the CoM position of the i^{th} link. Other potential energy, such as springs, can be added here as well.

The kinetic energy can be calculated as follows:

$$\begin{aligned} T_{i_{linear}} &= \frac{1}{2} \times m_i \times \dot{\mathbf{P}}_i \times \dot{\mathbf{P}}_i' \\ T_{i_{rotational}} &= \frac{1}{2} \times \omega_i' \times J_{i_{MoI}} \times \omega_i \\ T &= \sum (T_{i_{linear}} + T_{i_{rotational}}) \end{aligned} \tag{1.7}$$

where T is the total kinetic energy of the system, calculated by summing all the linear and rotation kinetic energies for each link. $T_{i_{linear}}$ is the linear kinetic energy for the i^{th} link, with $T_{i_{rotational}}$ being the rotational kinetic energy for the link. $J_{i_{MoI}}$ is the moment of inertia matrix for the i^{th} link, with ω being the **absolute** angular velocity of the link. If you are using relative coordinates, you will have to sum the angular velocities to get the absolute angular velocity of the link (thereby making the dynamics more complex).

1.3 Generating the Manipulator Equation

Once the energy of the system (T and V) have been calculated, the manipulator equation (1.1) can be generated.

To calculate the mass matrix, the following formula is used:

$$\mathbf{M} = jacobian(jacobian(T, \dot{\mathbf{q}})', \dot{\mathbf{q}}) \tag{1.8}$$

Using partial differentiation, the gravitational potential matrix is calculated as follows:

```
for i = 1:length(q)
   G(i) = diff(V,q(i))
end
```

Calculating the Coriolis matrix is slightly more complex and is done as follows:

```
for i = 1:length(q)
 for j = 1:length(q)
  for m = 1:length(q)
   temp=0.5(diff(M(i,j),q(m)) + diff(M(i,m),q(j)) - diff(M(j,m),q(i)))q̇(m);
   C(i,j) = C(i,j)+ temp;
  end
 end
end
```

The only thing left to calculate are **B** and **A**. **A** maps external forces, such as GRF, to the generalized coordinates.This is done using the Jacobian of the position of the external force (for example, GRF will act at the robots foot). This was done as follows:

$$\begin{aligned} \mathbf{A}_i &= jacobian(\mathbf{P}_{force_i}, \mathbf{q}) \\ \mathbf{A} &= \sum \mathbf{A}_i \end{aligned} \tag{1.9}$$

where $\mathbf{P}_{force_i}$ is the position of the i^{th} external force acting on the robot, and $\mathbf{A}_i$ is the **A** matrix component for the i^{th} external force.

Lastly the **B** matrix needed to be calculated. This matrix maps the actuators forces and torques to the appropriate generalised coordinates. This matrix changes if you use absolute or relative coordinates.

$$\mathbf{B} = jacobian(\theta, \mathbf{q}) \tag{1.10}$$

where θ is a vector, with each element being the relative angle or length the corresponding torque/force actuator acts on.

1.4 The B Matrix in more detail

Often $\mathbf{B}\boldsymbol{\tau}$ is jointly referred to as the vector **Q**. This sometimes makes it easier to generate the vector as apposed to the above method.

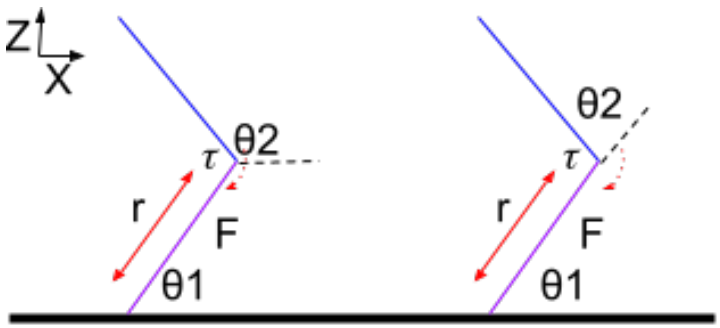

Figure 1.2: A two link pendulum. There is a torque actuator linking the first and second link, with a force actuator that can lengthen and shorten the first link. Absolute coordinates are on the left, relative on the right.

For example, if you have the system shown in Figure 1.2, with $q = [r, \theta_1, \theta_2]'$.
For relative angles (right):

$$Q = [F, 0, \tau]' \tag{1.11}$$

For absolute angles (left):

$$Q = [F, -\tau, \tau]' \tag{1.12}$$

as the relative angle where the motor acts is $\theta_2 - \theta_1$, taking $jacobian(relAngle, q)$ results in [0,-1,1].

1.5 Examples

The EoM for the two link pendulum shown in Figure 1.1 will be calculated. First we need to define the generalized coordinates:

$$\mathbf{q} = [\theta_1, \theta_2]' \tag{1.13}$$

1.5.1 Absolute Angles

The position for the CoM of each link is calculated as follows:

$$\begin{aligned} P1 &= R(\theta_1)[l1/2, 0, 0]' \\ P2 &= R(\theta_1)[l1, 0, 0]' + R(\theta_2)[l2/2, 0, 0]' \end{aligned} \tag{1.14}$$

This results in the following potential energy:

$$\begin{aligned} V_1 &= m_1 \times g \times P1 \\ V_2 &= m_2 \times g \times P2 \\ V &= V_1 + V_2 \end{aligned} \tag{1.15}$$

and kinetic energy:

$$\begin{aligned} T_{1_{linear}} &= \frac{1}{2} \times m_1 \times \dot{P}_1 \times \dot{P}_1' \\ T_{1_{rotational}} &= \frac{1}{2} \times [0, \dot{\theta}_1, 0] \times J_1 \times [0, \dot{\theta}_1, 0]' \\ T_{2_{linear}} &= \frac{1}{2} \times m_2 \times \dot{P}_2 \times \dot{P}_2' \\ T_{2_{rotational}} &= \frac{1}{2} \times [0, \dot{\theta}_2, 0] \times J_2 \times [0, \dot{\theta}_2, 0]' \\ T &= T_{1_{linear}} + T_{1_{rotational}} + T_{2_{linear}} + T_{2_{rotational}} \end{aligned} \tag{1.16}$$

where $\dot{P}$ can be calculated using the Jacobian as shown above and J is the relevant moment of inertia matrix of the link.

The relevant matrices (**M**, **C** and **G**) can then be calculated using the equations above. Assuming there is a torque actuator that joins the red and blue link, with the red link free to pivot at θ_1, the **Q** matrix becomes:

$$\mathbf{Q} = [-\tau, \tau] \tag{1.17}$$

as the relative angle (where the motor acts) is $\theta_2 - \theta_1$.

1.5.2 Relative Angles

The position for the CoM of each link is calculated as follows:

$$\begin{aligned} P1 &= R(\theta_1)[l1/2, 0, 0]' \\ P2 &= R(\theta_1)[l1, 0, 0]' + R(\theta_1 + \theta_2)[l2/2, 0, 0]' \end{aligned} \tag{1.18}$$

This results in the following potential energy:

$$\begin{aligned} V_1 &= m_1 \times g \times P1 \\ V_2 &= m_2 \times g \times P2 \\ V &= V_1 + V_2 \end{aligned} \tag{1.19}$$

and kinetic energy:

$$\begin{aligned} T_{1_{linear}} &= \frac{1}{2} \times m_1 \times \dot{P}_1 \times \dot{P}_1' \\ T_{1_{rotational}} &= \frac{1}{2} \times [0, \dot{\theta}_1, 0] \times J_1 \times [0, \dot{\theta}_1, 0]' \\ T_{2_{linear}} &= \frac{1}{2} \times m_2 \times \dot{P}_2 \times \dot{P}_2' \\ T_{2_{rotational}} &= \frac{1}{2} \times [0, \dot{\theta}_1 + \dot{\theta}_2, 0] \times J_2 \times [0, \dot{\theta}_1 + \dot{\theta}_2, 0]' \\ T &= T_{1_{linear}} + T_{1_{rotational}} + T_{2_{linear}} + T_{2_{rotational}} \end{aligned} \tag{1.20}$$

where $\dot{P}$ can be calculated using the Jacobian as shown above and J is the relevant moment of inertia matrix of the link.

The relevant matrices (**M**, **C** and **G**) can then be calculated using the equations above. Assuming there is a torque actuator that joins the red and blue link, with the red link free to pivot at θ_1, the **Q** matrix becomes:

$$\mathbf{Q} = [0, \tau] \tag{1.21}$$

Chapter 2

Trajectory Optimisation

Trajectory optimisation is a mathematical tool used to generate a trajectory that minimises (or maximizes) a measure of performance (a cost function) while satisfying a number of constraints and variable bounds. In simpler english, it is a search problem that is used to find a set of results that meet a set of constraints and bounds. The trajectory of each variable is divided into N node points. Constraints, such as the EoM, are applied to ensure a feasible transition from node to node. Constraints were satisfied by varying the decision variables between their respective bounds until a feasible or (locally) optimal solution was found.

Due to the complexity of the problems being optimised, there is no guarantee that the solver will find an optimal solution on the first attempt, often getting stuck in local minima. Therefore, to increase the likelihood that the solver will find an optimal solution, and to explore the search space, multiple optimisations are generally performed from a randomly generated seed points (the starting point of the optimiser) using uniformly distributed random numbers between the variable bounds. The optimal solution is then taken as the seed that resulted in the lowest cost value.

There are a number of different cost functions to choose from. As we are focused on rapid manoeuvres, some of the useful cost functions include:

$$\begin{aligned} J_1 &= \sum_{i=1}^{N} h(i) \\ J_2 &= \sum_{i=1}^{N} \frac{\tau(i)^2}{x_{final}} h(i) \\ J_3 &= \sum_{i=1}^{N} \tau(i)^2 h(i) \end{aligned} \tag{2.1}$$

where J_1 is minimum time cost function (where $h(i)$ is the time duration for the i^{th} node), J_2 is a cost of transport, CoT, cost function and J_3 is a torque (heat) based heat cost function. It must be noted that J_2 is not a true CoT and is rather the heat based cost function, J_3, normalized by the total distance travelled. However, it is proportional to the non-dimensional CoT cost (for electric actuators that suffer heat loss).

On a side note, we do not use weighted cost functions (for example $J = \alpha_1 J_1 + \alpha_2 J_2$), until someone can justify a suitable method of choosing those weights. It is a similar reason why I do not use the penalty method (see GRF chapter).

In order to guide the optimiser to a solution, a number of constraints and bounds are applied to the problem. Some of the common constraints and bounds are listed below.

2.1 Collocation

The trajectories of each generalised coordinate are discretised into N finite elements (time periods or node points, designated i) using polynomials. The trajectory between each node point is represented using a Runge-Kutta basis with 3 collocation points as seen in Figure 2.1, designated j. Therefore, the trajectory was divided into N points, ($i \in [1, N]$), each with 3 collocation points, ($j \in [1,3]$). Three-point Radau was used to solve the differential equations (EoM), at the selected time points. (On a side note: That is a lot of jargon that is not critical to understand if you are new to trajectory optimisation, basically it is saying we take a trajectory, cut it into N points, further divide it up according to the number of collocation points, then use polynomials to create a smooth trajectory). This was done by implementing the following equations for each state trajectory ($\mathbf{q}$ and $\dot{\mathbf{q}}$):

$$\begin{aligned}
\mathbf{q}(i,j) &= \mathbf{q}_0(i) + h(i) * \sum_{k=1}^{3} \mathbf{a}(k,j)\dot{\mathbf{q}}(i,k) \\
\dot{\mathbf{q}}(i,j) &= \dot{\mathbf{q}}_0(i) + h(i) * \sum_{k=1}^{3} \mathbf{a}(k,j)\ddot{\mathbf{q}}(i,k) \\
\mathbf{q}_0(i) &= \mathbf{q}(i-1,3) \\
\dot{\mathbf{q}}_0(i) &= \dot{\mathbf{q}}(i-1,3)
\end{aligned} \tag{2.2}$$

The first two equations implement the 3 point Radau to integrate the equations of motion from node i to $i+1$ whereas the third and fourth equation ensure that the third collocation point of node i was located at the same point as node $i+1$ to ensure a continuous state trajectory. The difference between $\mathbf{q}(i,j)$ and $\mathbf{q}_0(i)$ was that $\mathbf{q}(i,j)$ acts on i (node point) and j (collocation point), whereas $\mathbf{q}_0(i)$ only acts on i. The 3 point collocation matrix is as follows:

$$\mathbf{a} = \begin{bmatrix} 0.19681547722366 & 0.39442431473909 & 0.37640306270047 \\ -0.06553542585020 & 0.29207341166523 & 0.51248582618842 \\ 0.02377097434822 & -0.04154875212600 & 0.11111111111111 \end{bmatrix} \tag{2.3}$$

For systems that make use of contacts that are required to occur at node points (and not between node points), a variable time-step is required. This allows the optimiser to increase the resolution at critical points in the trajectory (for example contact events) and have a lower resolution at less critical points (aerial phases). The time-step between each finite element is constrained as follows:

$$\begin{aligned}
0.5h_M \leq h(i) &\leq 2h_M \\
tt(i,j) &= tt_0(i) + h(i) \sum_{k=1}^{3} \mathbf{a}(k,j) \\
tt_0(i) &= tt(i-1,3)
\end{aligned} \tag{2.4}$$

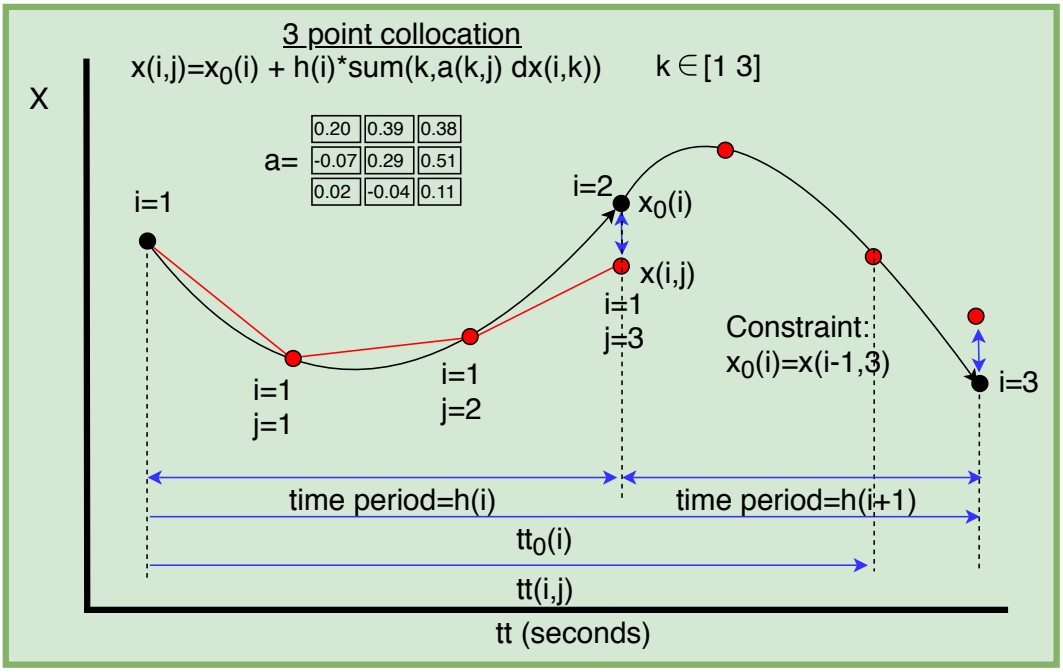

Figure 2.1: A graphical representation showing how the trajectory is first broken up into node points (i) and then collocation points (j). Additional constraints to ensure the trajectory is smooth are also shown.

where $h(i)$ was the time-step for the i^{th} node and h_M was set to T/N, where T is used as a scaling factor for the node time and was set according to the expected time the task would take. tt is the total time from the start of the trajectory to the relevant node point and collocation point. The first equation bounds the minimum and maximum time period for the node trajectory, whereas the second equation splits up the node time among the collocation points and the third equation ensures the start time of node i ($tt_0(i)$) matches the end time of node $i-1$ at the third collocation point ($tt(i-1,3)$).

2.2 Terminal Conditions

Generally start and end conditions are required, especially in robotics, and it helps to define the task that must be achieved (for example, start at rest, end at rest, having travelled a fixed distance). You can set both the start and end constraints using bounds or constraints, however you need to ensure you do not over constrain your problem.

Typically I set the start conditions using bounds, and the terminal conditions with constraints. If both are set with bounds, the optimiser does not have enough freedom to find a solution, if both are set with constraints, the optimiser has too much freedom.

2.3 Average Velocity

For robotic applications, you often want to enforce an average velocity (note that the initial and terminal velocities are not specified, only the average is constrained). This is enforced using the following constraint:

$$\dot{x}_{average} = \frac{x_0(N) - x_0(1)}{tt_0(N) - tt_0(1)} \tag{2.5}$$

where $x_0(N)$ is the final distance reached and $x_0(1)$ is the initial starting point ($x_0(1) = 0$). $tt_0(N)$ is the final time and $tt_0(1)$ is the starting time ($tt_0(1) = 0$). The constant $\dot{x}_{average}$ was the enforced average velocity.

2.4 Joint Angles

If relative angles are used, then you can constrain the range of motion using the upper and lower bounds on the variable. If absolute angles are used, constraints are required to enforced the range of motion and velocity of the links. If we take the example of a two pendulum in Figure 1.2, the constraints are as follows:

$$\begin{aligned} lower\ bound < \theta_2 - \theta_1 < upper\ bound \\ lower\ bound < \omega_2 - \omega_1 < upper\ bound \end{aligned} \tag{2.6}$$

Bounds can be used to limit the range of θ_1 and ω_1 as it is the same angle for relative and absolute angles.

2.5 Motor Model

In order to make the results applicable to robotic platforms, a linear motor power model is often implemented, taking into account the stall torque and no load speed of the specified motors. This is implemented through constraints as follows (bearing in mind that relative velocities are used):

$$-\tau_{max} - \frac{\tau_{max}}{\omega_{max}}\omega(i) \leq \tau(i) \leq \tau_{max} - \frac{\tau_{max}}{\omega_{max}}\omega(i) \tag{2.7}$$

where τ_{max} is the stall torque of the motor and ω_{max} is the no load speed of the motor.

2.6 Time Upper Bound

In order to enforce a rapid manoeuvre, an upper bound can be placed on the total time allowed for the trajectory. This is implemented using the following constraint:

$$\sum_{i=1}^{N} h(i) < T_{max} \tag{2.8}$$

The upper bound on time, T_{max}, is generally hand chosen, by manually reducing it until the optimiser could no longer find a feasible solution.

2.7 Hard Stops for Prismatic Actuators

For the prismatic actuators, hard stops are implemented to stop the actuator from over extending or over contracting. These hard stops are implemented using complimentary constraints (similar to the GRF, see later chapter) as follows (for the minimum length):

$$\begin{aligned} \alpha_{HS}(i,j) &= l(i,j) - l_{min} \\ \beta_{HS}(i,j) &= F_{HS}(i,j) \\ \alpha_{HS}(i,j)\beta_{HS}(i,j) &= 0 \end{aligned} \tag{2.9}$$

For the maximum length hard stop constraint, the following formulas are used:

$$\begin{aligned} \alpha_{HS}(i,j) &= l_{max} - l(i,j) \\ \beta_{HS}(i,j) &= F_{HS}(i,j) \\ \alpha_{HS}(i,j)\beta_{HS}(i,j) &= 0 \end{aligned} \tag{2.10}$$

2.8 Bounds

To restrict the problem size, variable bounds are enforced on all the decision variables. The optimiser varies these variables between their bounds to find a trajectory that satisfies the above constraints and minimises the specified cost function.

Often bounds are placed on GRF to ensure that excessively large forces are not experienced, which can lead to breaking legs. The ground reaction forces, $\boldsymbol{\lambda}$, are typically bounded to five times the mass of the robot, $5m_{robot}g$, where g was the gravity constant, 9.81 m/s^2.

All initial conditions are enforced using bounds (whereas the terminal conditions were enforced through constraints) by setting the upper and lower bound of the first node to the desired value.

The remaining variables that are not bounded through constraints are bounded sufficiently high as to not bias the solution but to restrict the size of the problem.

2.9 Tricks

2.9.1 1 or 0 variables (bang-bang)

2.9.2 Absolute value function

Chapter 3

Through Contact Methods

In order to explore the search space properly, it is a bad idea to enforce a gait pattern. Therefore, through contact methods are implemented to allow the optimiser to pick the optimal gait pattern. These methods also allow the foot to slip/make multiple contacts (think ABS breaking). Previous studies using this method have shown that it is a vital tool in studying complex locomotion. There are some downfalls to this method, such as requiring a large number of complimentary constraints to ensure that the GRF only acts when the foot is on the ground. For this course, only two dimensional contacts are investigated in detail. Dean has developed a method for 3D contacts which will be touched on later.

In order to implement 2D through contact methods, the horizontal component of the ground reaction force at each foot/contact point is split into the positive, λ_x^+, and negative, λ_x^-, component as follows:

$$\boldsymbol{\lambda} = [\lambda_x^+ - \lambda_x^-, \lambda_z]' \tag{3.1}$$

with the following positive variables (duplicated for each contact point):

$$\lambda_z, \lambda_x^+, \lambda_x^- \geq 0 \tag{3.2}$$

First the friction cone is enforced using the following constraints:

$$\mu\lambda_z - \lambda_x^+ - \lambda_x^- \geq 0 \tag{3.3}$$

with the coefficient of friction, μ, being typically being set to 1 for all simulations (you can google a better value if you want, most research limits it to 1). The magnitude of the relative tangential velocity at the contact is calculated with the following constraints:

$$\begin{aligned} \gamma + \Psi(\mathbf{q}, \dot{\mathbf{q}}) &\geq 0 \\ \gamma - \Psi(\mathbf{q}, \dot{\mathbf{q}}) &\geq 0 \\ \gamma &\geq 0 \end{aligned} \tag{3.4}$$

where γ is the magnitude of the relative velocity and $\Psi(\mathbf{q}, \dot{\mathbf{q}})$ is the relative tangential velocity. This requires that γ is a positive variable (third equation in the above set of equations). In order to ensure that the ground reaction force is only applied to the foot when it was on the ground, the following complementarity constraint are applied (shown in Figure 3.1):

$$\phi(\mathbf{q})^T \lambda_z = 0 \tag{3.5}$$

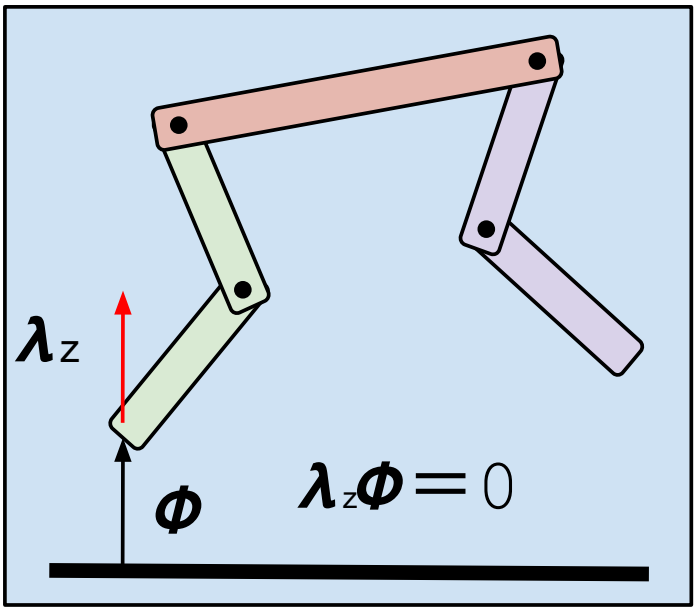

Figure 3.1: Image showing an example complementarity constraint. Note how the foot height, multiplied by the vertical GRF must equal zero. Therefore, if the foot is in the air, the GRF must be zero and vice versa.

where $\phi(\mathbf{q})$ is the vertical height of the foot (positive by nature). For this equation to be satisfied, a ground reaction force can only be applied when the foot is on the ground and must be zero if the foot is not on the ground. In order to ensure that the friction force lies on the friction cone when the foot slides, the following complementarity constraint is applied:

$$(\mu\lambda_z - \lambda_x^+ - \lambda_x^-)^T \gamma = 0 \tag{3.6}$$

Finally, the following two complementarity constraints are applied to ensure that the horizontal ground reaction force opposes the sliding motion of the foot (by using the absolute value of the tangential velocity of the food to turn on/off the horizontal components of friction) :

$$\begin{aligned}(\gamma + \Psi(\mathbf{q}, \dot{\mathbf{q}}))^T \lambda_x^+ = 0 \\ (\gamma - \Psi(\mathbf{q}, \dot{\mathbf{q}}))^T \lambda_x^- = 0\end{aligned} \tag{3.7}$$

The above constraints need to be duplicated for each contact point (this is why we model feet as points!). All contacts in this method are modelled as inelastic collisions and can have only one of two modes: slipping or sticking. Slipping is modelled using a simple coulomb friction model.

Through contact methods are notoriously difficult to solve, as they contain a large number of complimentary constraints. Therefore to further improve convergence times,

regularisation methods in the form of ϵ relaxation techniques are employed to solve complementarity constraints ($\alpha\beta = 0$) from the through-contact optimisation methods. This was done by re-phrasing the above constraints as follows:

$$\alpha\beta \leq \epsilon \tag{3.8}$$

where α and β form the two parts of the complimentary constraints. Initially ϵ is set to 1000 and the problem is solved iteratively 8 times. After each successful solve, ϵ is divided by 10. As soon as a solution is not found, the next seed point is run. If it succeeded 8 times, then the complementarity constraints were considered solved (within an acceptable tolerance, $\epsilon = 1E-4$) and the solution is saved.

In order to further simplify the problem and increase the convergence rate, these complementarity constraints are summed across the collocation points (j) and only evaluated at node points (i) as follows:

$$\begin{aligned} \alpha'(i)\beta'(i) &\leq \epsilon \\ \alpha'(i) &= \sum_{j=0}^{K} \alpha(i,j) \\ \beta'(i) &= \sum_{j=0}^{K} \beta(i,j) \end{aligned} \tag{3.9}$$

Once solved, the complementarity constraint will equal zero ($\alpha(i,j)\beta(i,j) = 0$, within the tolerance of the optimiser and the value of ϵ).

3.1 Epsilon relaxation VS penalty method

In the literature there are two methods to solving complementarity constraints. The prefered method is the epsilon relaxation method shown above. The second commonly used method is called the penalty method. The penalty method makes the following modifications to the above complimentary constraints

$$\begin{aligned} \alpha_1\beta_1 &= penalty_1 \\ \alpha_2\beta_2 &= penalty_2 \\ &\ldots \\ J &= cost + \sum(weight_1 penalty_1 + weight_2 penalty_2 + ...) \end{aligned} \tag{3.10}$$

As can be seen, you multiply out the complimentarity constraints and save them as a positive variable. In the cost function, you then sum these variables (if the complimentarity is solved, they should equal zero) and multiply them by a large value known as a weight and add it to the cost function.

The optimizer tries to minimize the cost function, therefore by solving the complimentarity constraints, the weight x penalty will tend to zero and then the objective function can be solved. There are a number of reasons I hate this method and refuse to use it. These reasons include:

- How do you select your weights?

- The optimizer spends most of its time solving the complimentarity constraints and not the objective function. So if you get an optimal solution, it may not be optimal in terms of your desired cost function. Its the same reason why you don't have multiple objectives in your cost function (minimum time and minimum energy, they have different units as well)
- How do you know the complimentarity constraints are actually solved? (you can get an optimal solution with a large objective function, whereas the epsilon method, if it solves then its guaranteed that the complimentarity constraints are solved)
- The optimiser can scale the objective function, so your weights are pointless.

If you do use the penalty method, you absolutely need to perform post processing to ensure the complimentarity constraints are satisfied. I would only use this method to generate a seed point for the epsilon relaxation method.

3.2 3D contacts-Dean's Method

Chapter 4

Solving Methods

In order to improve the convergence rate and the time taken to converge, a number of heuristics can be employed. One such method commonly employed involves only randomizing the initial values of the generalised coordinates ($\mathbf{q}$ and $\dot{\mathbf{q}}$ and fixing the remaining variables of the seed to 0.01. Another method used is to solve the seed iteratively, each iteration adding more constraints until all the constraints have been added (with the solver tolerances being set low, $tol = 1E-4$, and the cost function fixed to a constant value). Once all the constraints have been added, the solver can be warm started with tighter tolerances, $tol = 1E-6$, and the desired cost function.

Chapter 5

Using Jupyter Notebook

There is available code for Jupyter for a simple monopod hopper available here:
`https://github.com/callenFisherLeggedRobotics/PyomoTutorials`

5.1 Getting Started

- The tutorial that I followed to get started:
  `https://jckantor.github.io/ND-Pyomo-Cookbook/`
- First you need to install Anaconda:
  `https://www.anaconda.com/products/individual`
  I used the 64 bit graphical installer for Python 3.7
- Install any available package updates. Open a Anaconda Prompt command line, and execute the following two commands on separate lines.
  conda update conda
  conda update anaconda
- Then install Pyomo
  conda install -c conda-forge pyomo
  conda install -c conda-forge pyomo.extras
- Install solvers
  conda install -c conda-forge ipopt
- Open up Jupyter notebooks
  find the correct path, then: new-python 3 or open the SS_mono file and run it

Chapter 6

Using Google Colab

6.1 Getting Started

6.2 Running a simple model