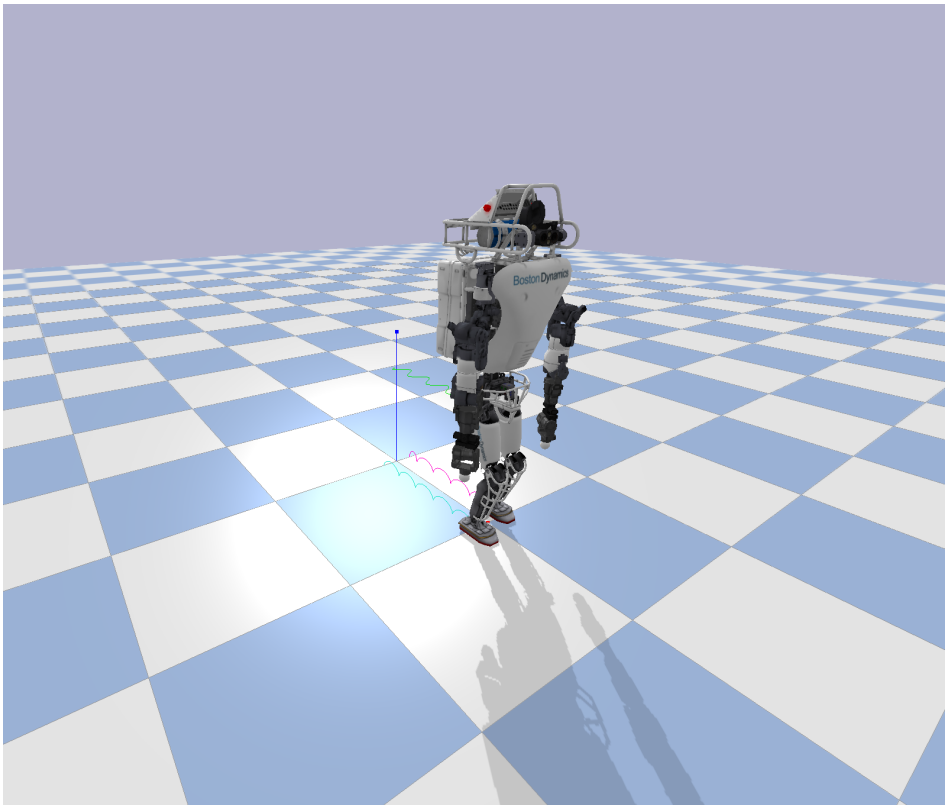
EPFL

MICRO-507 - Legged Robots
Autumn Semester 2021

Report on Project 1 :

Locomotion Planning Based on Divergent Component of Motion



Benjamin Mottis
Antoine Perrin
Matteo Righi

Contents

1 Introduction

The goal of this project is to implement a trajectory planner for a biped robot using divergent component of motion (DCM), as presented in [1]. The trajectory planner is written in Python and then implemented in PyBullet to control an Atlas robot. Tweaking the parameters of the simulation, it is possible to vary the gait of the robot. The completed implementation can be compared with a previous assignment in Matlab, notably to compare the cost of transport.

2 Methods

The principle of DCM control is to plan the locomotion of the robot by first planning the trajectory of the divergent component of motion.
The code includes three blocks: a DCM planner, a foot trajectory planner, and an inverse kinematics block. Only the first, the DCM planner, needs to be completed.

2.1 Summary of the method for the DCM trajectory calculation

We present here a summary of the calculation to plan the trajectory of the DCM, as given in the assignment document:
The movement of the legs can be modelled using the linear inverted pendulum (LIP) model.
The DCM is defined as the quantity:

$$\xi = x_c + \frac{\dot{x_c}}{\omega} \tag{1}$$

$$\xi(t) = r_{cop} + (\xi_0 - r_{cop})e^{\omega t} \tag{2}$$

We will calculate the trajectory of the DCM for a desired movement velocity as follows:

1. The position of each footstep and the step duration are set based on the desired velocity (in fact, these must be tuned for the desired velocity to produce a stable movement). The centre of pressure (CoP) at each footstep $\{r_{CoP,1}, ..., r_{CoP,N}\}$ is placed so as to ensure dynamic balance

2. We define $\xi_{ini,i}$ the position of the DCM at the beginning of step i, and likewise $\xi_{EoS,i}$ the position of the DCM at the end of the step.
   Then we impose that the DCM at the end of the last step $\xi_{EoS,N-1}$ is on the corresponding CoP to make the robot stop at this step

3. The DCM for every other end of step (and for the beginning of the first step) can then be deduced recursively:
   $$\xi_{EoS,i-1} = \xi_{ini,i} = r_{CoP,i} + (\xi_{EoS,i} - r_{CoP,i})e^{-\omega T} \tag{3}$$
   Where T is the step duration

4. The position of the DCM at every point in between each end-of-step can be calculated from equation 2:
   $$\xi_i(t) = r_{CoP,i} + (\xi_{EoS,i} - r_{CoP,i})e^{\omega(t-T)} \tag{4}$$

5. To prevent discontinuities at the transition between steps, we give the robot a double support phase (ie where it is supported by both legs). We define this double support phase to take a given time. Then we define $\{\xi_{iniDS,i}\}$ and $\{\xi_{EoDS,i}\}$ the positions of the DCM at the extremities of each double support phase (ie where the double support phase overlaps with the single support phase).
   We assume that the trajectory of the DCM during the double support phase has the form of a cubic polynomial. We can calculate it by interpolation from the position and the velocity of the DCM at the extremities of the double support phase, calculated in the previous point.

2.2 Implementation in Python

The DCM trajectory generator is implemented in Python by the class `DCMTrajectoryGenerator()`.

Firstly, we will generate the DCM trajectory considering only single support. To do this, the CoM trajectory is calculated using the function `getCoMTrajectory()` which finds the trajectory of the CoM by integrating its velocity, as given by equation 1. The function `findFinalDCMPositionsForEachStep()` is completed using equation 3. Then, the trajectory of the CoP is computed using the function `calculateCoPTrajectory()` in which $\dot{\xi}$ is calculated by deriving ξ. Finally, the DCM is computed at each iteration of the simulation using the function `planDCMForSingleSupport()`.

Next, we modify the single-support DCM trajectory to contain double-support parts; this will give the robot a smoother and more stable trajectory. It was necessary to define the boundary conditions for double support with the function `findBoundaryConditionsOfDCMDoubleSupport()`, then the coefficients of interpolation are computed in the function `doInterpolationForDoubleSupport()`. Finally, the DCM double support trajectory is computed at each step of the simulation in the function `embedDoubleSupportToDCMTrajectory()`.

3 Results

3.1 Program Results

Once the Python script was complete, we were able to produce satisfactory step plots, but the robot was not able to balance well. To generate a stable trajectory we tuned the parameters; the details for the tuning are explained in section 4, parts 4 and 6. Using these parameters, we produced the graphs shown in figures 1, 2, 3.

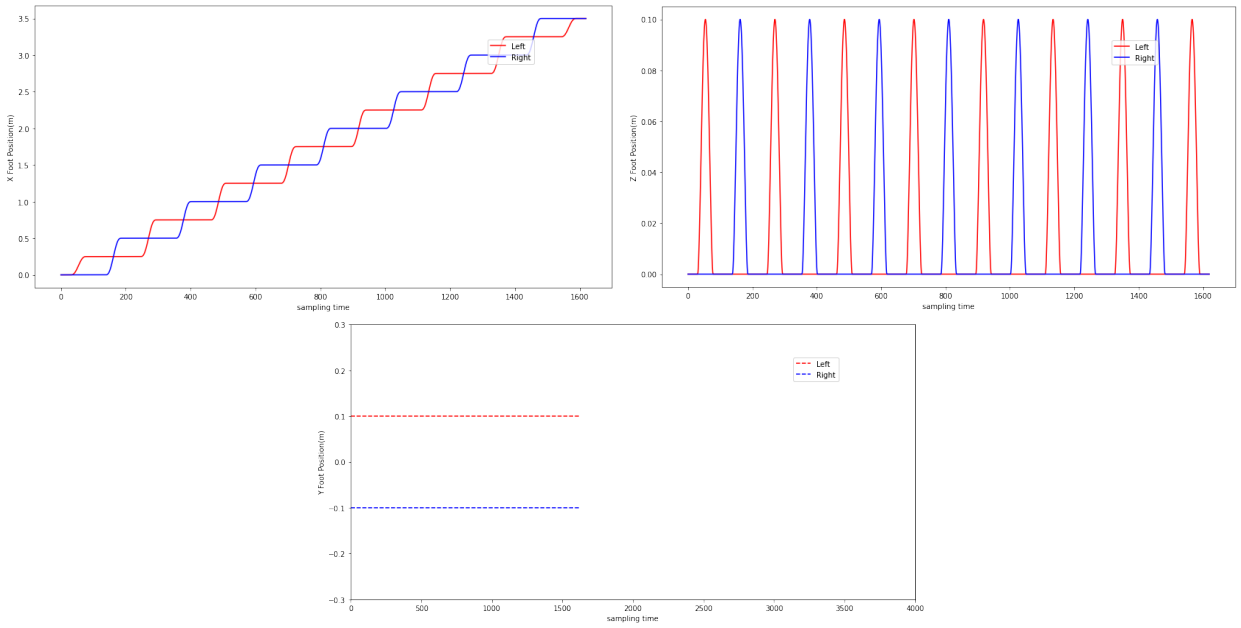


Figure 1: x-, y- and z-trajectories of the robot's foot

We can see, in particular, that the DCM and the COM are stable. We used these plots to compute the general speed of the robot and to tune the different parameters.

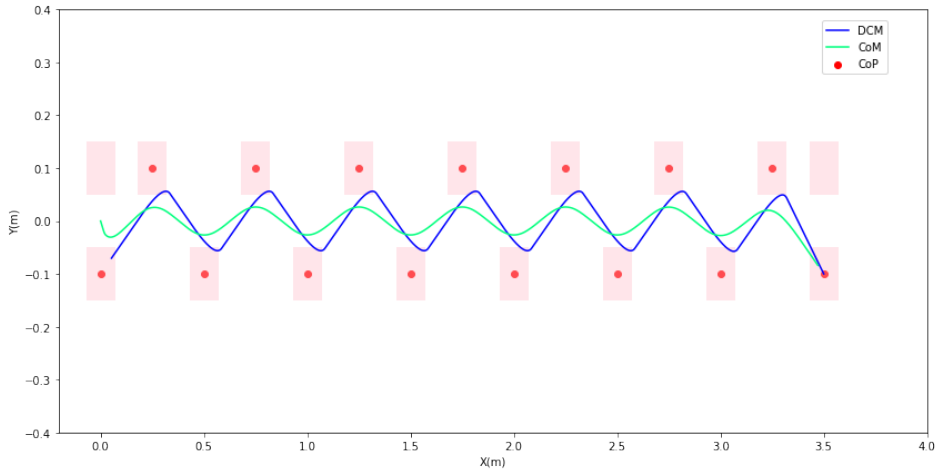


Figure 2: Foot, DCM and COM trajectories in 2D space

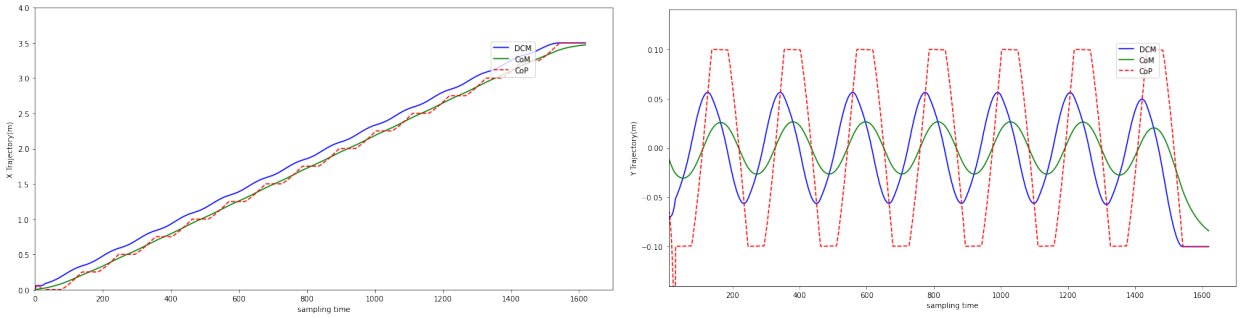


Figure 3: x- and y-trajectories of the foot, DCM and COM in time

3.2 Discussion on the CoT: Comparison with the Matlab Simulation

In the following section, we will calculate the cost of transport (CoT) for both the Matlab simulation from the previous assignment and the Python simulation from this assignment. For both, we calculate an instantaneous CoT at each point in time:

$$CoT(t) = \frac{P(t)}{mgv(t)} \tag{5}$$

We will then compare the mean of the absolute-value of the CoT for both.

3.2.1 Explanation of the CoT formulas

For the Matlab assignment, the final expression to calculate the cost of transport CoT which we used is:

$$CoT = \frac{P_{movement}}{m_{tot}gv} = \frac{\Sigma_i u_i \dot{\theta}_i}{(\Sigma_i m_i)gv_h} = \frac{u_1(\dot{q}_1 - \dot{q}_2) + u_2(\dot{q}_2 - \dot{q}_2)}{(m_1 + m_2 + m_3)g\dot{x}_h} \tag{6}$$

We arrive at this formula using:

- The power provided to the system is the power to produce the torques u_1, u_2 on the two actuators of the robot. We can reference the assignment document for the Matlab assignment to calculate the angles θ_1, θ_2 on which the torques are producing a velocity:

$$\begin{cases} \theta_1 = \pi + q_1 - q_3 \\ \theta_2 = \pi + q_2 - q_3 \end{cases} \tag{7}$$

$$\Rightarrow \begin{cases} \dot{\theta}_1 = \dot{q_1} - \dot{q_3} \\ \dot{\theta}_2 = \dot{q_2} - \dot{q_3} \end{cases} \tag{8}$$

- The mass which is moved is the total mass of the biped, in this case the sum of the point masses of each limb:
$$m_{tot} = m_1 + m_2 + m_3 \tag{9}$$
- We used for the velocity the x-direction velocity of the hip joint: $v = \dot{x}_h$.
We chose to use the velocity of the hip joint because the hip joint does not oscillate the way the legs do; it is also analogous to the centre of mass, which we use in the Python model. We chose to use only the x-component of the velocity because CoT is a measure of efficiency, and in this case the desired work output is movement in the x direction. This corresponds also to the equivalent formulation of the CoT using the distance travelled d, which in this case would be in the x direction:
$$CoT = \frac{E}{mgd} = \frac{P}{mgv} \Leftrightarrow v = \frac{dd}{dt}$$

For the Python assignment, the Cot is calculated similarly:

$$CoT = \frac{\Sigma_i u_i \dot{q_i}}{mgv_x} \tag{10}$$

We arrive at this formula using:

- As in the previous case, the power provided to the system is the power to produce the motor torques. In this case, the angles corresponding to each motor can be extracted from the code without additional calculations.
Note that unlike in the Matlab assignment, in this assignment the Atlas robot has 28 actuators [2]; we are considering for the CoT the power produced by each actuator.
- We consider the mass to be the total mass of the robot of 89kg [2].
- We consider for the velocity only the velocity in the x-direction. This is coherent with what we used in the Matlab implementation, and the reasoning for it is the same.
The velocity is calculated in the Python script by numerically differentiating the array `ComTrajectory`.

3.2.2 Choosing the parameters

First, the parameters of the Python code are tuned arbitrarily to obtain a stable gait on 17 steps:

```
doubleSupportDuration = 0.1
stepDuration = 0.6
pelvisHeight= 0.75
maximumFootHeight = 0.07
stepWidth=0.12
stepLength= 0.1
numberOfFootPrints=17
```

We can then run the PyBullet simulation; we find that the robot moves at a mean speed of 0.155 m/s.

Recall that the Matlab simulation uses a PID controller which maintains the torso at a given lean angle and the legs at the same angle from the ground; the parameters of the controller, as well as the initial condition of the robot, are determined so as to minimize a given cost function; in our case:

$$cost = |\dot{x}_h - v_{target}|$$

Where v_{target} is the desired movement velocity; we will set this to the velocity of the Python robot $v_{target} = 0.155 m/s$, so that the Matlab simulation is as close as possible to the Python simulation.

Note that we use for the masses and lengths of the robot in the Matlab simulation the default parameters $m_1 = m_2 = 7kg, m_3 = 17kg$ and $l_1 = l_2 = 0.5m, l_3 = 0.35m$. Ideally, the masses and lengths of the robot in the Matlab simulation should be matched to those of the Atlas robot: height of 1.5m and weight of 89kg [2]. However, with these parameters, the optimizer in the Matlab simulation cannot find a stable solution.
We will assume that it is still fair to compare the CoT even with these masses and lengths because the two robots are still of similar height, and because the difference in mass has a less important effect on the CoT, as the CoT accounts for mass.

3.2.3 CoT Calculation, Comparison with the Matlab

Figure 4 shows the instantaneous CoT at every timestep computed in Matlab and in Python (note that the peaks have been cropped out for better legibility):

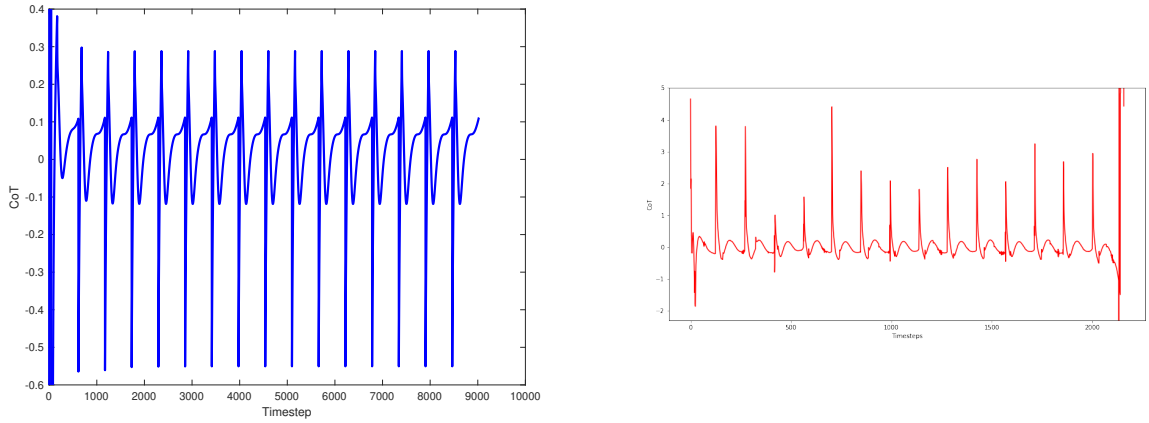


Figure 4: CoT in Matlab (blue) and Python (red)

To compare the two implementations, we will compare the mean of the absolute value of the CoT. Indeed, the Cot can be theoretically negative if the motor is slowing down the joint; however, this still consumes power, therefore it has a positive associated cost.
Furthermore, we can see in both the Matlab and in the Python that some steps have a much higher CoT than the others: in Matlab the first step, and in Python the first and last one. These skew the average CoT, so it might be more "fair" to compare the CoT of the two in steady-state only. To do this, we will remove the "outlier" steps from the two CoT vectors.

Using this methodology, we obtain: in Matlab an average absolute-value CoT of 0.088 and in Python of 0.216.

The two different assignments give different costs of transport because they correspond to radically different models. In particular the Matlab model is two-dimensional, while the Python model is 3-dimensional and uses a much more complicated robot; the additional constraints that this causes might explain the increased CoT.

Note that it is possible in the Matlab simulation to explicitly constrain the optimizer to choose a solution with a lower CoT, for example by using a cost function of the form:

$$cost = w_1|\dot{x}_h - v_{target}| + w_2|\bar{CoT}|; w_1, w_2 \in \mathbb{R}^*$$

Using this cost function reduces the CoT, however, it will tend to drive the robot faster, thus the comparison with the Python simulation would no longer be fair.

4 Answers to the Questions

1. Equation 2 (equation (5) in the assignment document) describes the analytical expression for the DCM. This shows that the DCM moves away from the centre of pressure in an exponential fashion. How fast it moves away, ie the rate of divergence, depends on the natural frequency of the oscillator $\omega = \sqrt{\frac{g}{z_c}}$, where z_c is the height of the centre of mass above the ground, and on the distance $\xi_0 - r_{CoP}$ between the initial position of the DCM at the beginning of the step and the CoP.

2. In DCM planning motion, the dynamic balancing conditions is guaranteed by setting the next foot step to ξ at the beginning of a each step and therefore the velocity $\dot{x}$ tends to 0 as shown in the 2 equations below:

$$\begin{aligned} \dot{\xi} &= \omega * (\xi - x_{base}) \\ \dot{x} &= -\omega(x - \xi) \end{aligned} \tag{11}$$

   In addition, as specified in the assignment document, the desired CoP is placed in a fixed location inside the foot print to guarantee dynamic balance during locomotion.

3. The DCM is based on the Linear Inverted Pendulum (LIP) model. In this model, the robot is represented as a point mass which remains at a constant height on a telescopic leg. Because of this simplification, even though dynamic balancing is guaranteed, it is necessary to properly tune the parameters of the robot in order to make it walk in a stable fashion.

4. In order to achieve stable locomotion we tuned the step duration and the double support duration. The pelvis height and the maximum foot height are also important for equilibrium; in our case, the default values were already suitable to generate stable locomotion.

   We reduced the step duration to avoid having the robot on a single foot for too long, to increase the stability of dynamic balancing.

   We observe that when the double step duration is increased, the trajectory of the DCM becomes less angular; we found that when the curve is too angular, the robot will tend to fall on the outward side (on the side of the stance leg), while if it is too smoothed the robot may fall on the inner side. We tuned this parameter to reach the correct balance of the two.

   The parameters which we used are:
   `doubleSupportDuration = 0.1`
   `stepDuration = 0.6.`
   `pelvisHeight= 0.7`
   `maximumFootHeight = 0.07`

5. Discussed in section 3.2.

6. To maximise the speed of the gait, we reduce the step duration and increase the step length as much as possible, to make the robot doing fasts larges steps.

   The step width was tuned for out chosen speed: we found that when the step width is too wide, the robot tends to swing from side-to-side, and when the step width is too narrow the robot tends to fall outwards.

   The pelvis height is slightly reduced, to lower the centre of mass and therefore to gain stability. This also increases the maximum step height, which can help to prevent the robot from stumbling.

   This gives the following parameters:
   `doubleSupportDuration = 0.25`
   `stepDuration = 0.45`
   `pelvisHeight= 0.65`
   `maximumFootHeight = 0.1`
   `stepWidth=0.1`
   `stepLength= 0.25`
   With these parameters, we obtain a speed of 0.55 m/s.

5 Conclusions

We are happy with the results of this project: the code appears to function correctly, and the results are as we expected.

Regarding the CoT computation, it is possible that the difference in mass and height between the two models had an effect; this is something that could be improved in another version of this project. Note, however, that in both cases, the measure is not very realistic, as it does not take into account losees in the actuators or losses due to friction, uneven terrain, etc. Another element to consider is that this comparison was for an arbitrarily chosen speed; it might have been more interesting to compare the the CoT-optimal speed (the speed which minimizes the CoT).

Overall, this project has been a useful exercise of programming in Python and in Matlab, as well as a good way to understand theoretical notions from the lectures.

6 References

[1] Englsberger, Johannes, Christian Ott, and Alin Albu-Schäffer. *Three-dimensional bipedal walking control based on divergent component of motion.* IEEE Transactions on Robotics 31.2 (2015): 355-368.(254)

[2] Boston Dynamics webpage for the Atlas robot: `https://www.bostondynamics.com/atlas` ; accessed on 14/11/2021.