Dynamic Locomotion of Simplified Hopping Robots

Helena Hauter, Florian Leitner, Leonie Freisinger

Technical University of Munich, 31. March 2021

I. Project Objective

For achieving an agile movement, legged robots need to be capable of walking and running. For the spring-mass like dynamics of running a leg can be approximated by a single spring leg [10]. This helpful approach is represented by the spring-loaded inverted pendulum (SLIP) model. It abstracts spring-like leg behavior in a simple and very similar way to human running and has the advantage of being dynamically stable for single-leg hopping and walking.
The goal of this project is to implement a SLIP running motion of an articulated robotic leg based on a paper of Marco Hutter [8]. This includes the projection of the SLIP model behavior onto the dynamics of an articulated robotic leg, implementing an applicable control structure, and the impact compensation at the entering of the stance phase.

II. Theory

A. SLIP Model

The SLIP model is described by a point mass m, which is attached to a mass-less spring with the resting length l_0 and the stiffness k, which represents the leg stiffness.

With this spring-loaded inverted pendulum, running can be approximated. In detail, the running motion is accomplished by alternating between a stance phase and a flight phase. The stance phase is entered as soon as the leg strikes the ground with the angle of attack α. In this phase the motion of the center of gravity of the SLIP model is directed by the spring force in the leg

$$\boldsymbol{F_{leg}} = k(l_0 - l) \tag{1}$$

that acts between the ground and the mass. This leads to the equation of motion,

$$m\ddot{\boldsymbol{r}}_{\boldsymbol{SLIP}} = \boldsymbol{F_{leg}} + m\boldsymbol{g} \tag{2}$$

where $\ddot{\boldsymbol{r}}_{\boldsymbol{SLIP}}$ describes the acceleration of the point mass.

The stance phase is over as soon as the leg force becomes equal to zero. In the flight phase only the gravitational force is acting on the model.

$$m\ddot{\boldsymbol{r}}_{\boldsymbol{SLIP}} = m\boldsymbol{g} \tag{3}$$

The SLIP model is energetically conservative and the apex height and forward speed are coupled to each other over the energy level.

B. Articulated Leg

As show in Fig. 1 the articulated leg is made of a floating base, two links and a foot. All segments are linked via rotational joints.
The robot dynamics are given by

$$\boldsymbol{M}\ddot{q} + b + g + J_s^T F_s = S^T \tau \tag{4}$$

For a better overview a table with short definitions of all used variables is given in I.

The overall goal is that the robotic leg behaves like the SLIP model. To achieve this, the stance dynamics of the robotic leg are projected onto its center of gravity. For a detailed derivation of the terms, please refer to [8]. For the articulated leg, only the hip and the knee joints are actuated. During the stance phase, an operational space controller is used to project the SLIP dynamics onto the articulated leg. For the flight phase, different PID controllers are used.
The operational space controller is given by

$$\boldsymbol{F} = \Lambda^* \ddot{r_{COG}} + \mu^* + p^* \tag{5}$$

To project the movement of the SLIP model onto the center of gravity of the leg $\ddot{r_{COG}}$ gets substituted by $\ddot{r_{SLIP}}$. To receive the torques that actuate the hip and knee joints the control law gets mapped into joint space. The resultant control law is then given by

$$\boldsymbol{\tau} = J^{*T}(\Lambda^* \frac{1}{m}(F_{leg} + mg) + \mu^* + p^*) \tag{6}$$

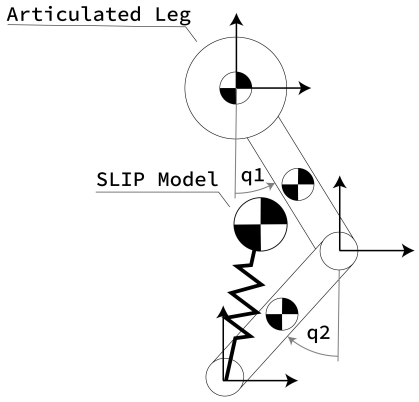

Fig. 1: Structure of the articulated Leg

Variable	Definition
q	state vector of the two segmented leg with q = [$x_{base}, y_{base}, \phi_{hip}, \phi_{knee}]^T$
τ	actuation torques
S	selection matrix (selects which parts are actuated)
J	Jacobian
M	mass matrix
b	Coriolis and centrifugal force vector
g	gravitational force
$< * >_s$	subscript s refers to the variables in support space
Λ^*	task inertia matrix
μ^*	projected Coriolis and centrifugal terms
p^*	projected gravitational term

TABLE I: Variable Definition

Because of the inelastic collision between floor and foot the robotic leg undergoes a velocity change that leads to the energy loss ΔE. To achieve continuous hopping equivalent to the SLIP model, impact compensation needs to be introduced. The lost energy gets stored as spring energy. Therefore the length of the SLIP model is changed, which results in a virtual pre-compression of the spring.

$$\mathbf{\Delta} E_{kinetic} = \Delta E_{spring} \tag{7}$$

$$\mathbf{0}.5k\Delta l^2_{SLIP} = 0.5m_{COG}(\dot{r}_{COG_{pre-impact}}{}^2 - \dot{r}_{COG_{post-impact}}{}^2) \tag{8}$$

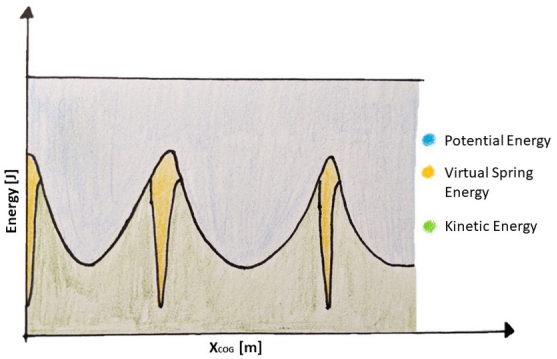

Fig. 2: Energy Diagram

The model thereby becomes energetically conservative. As shown in Fig.2 the total energy is constant for the whole hopping process. When the leg reaches the stance phase, the lost kinetic energy gets instantaneously shifted to spring energy. In the energy diagram, the robotic leg starts in the stance phase where the virtual spring energy is maximal, and the potential energy, as well as the kinetic energy, is minimal. As soon as the leg enters the flight phase, the spring energy is zero, and at apex height, the potential energy is maximal.

III. Implementation environment

To successfully implement the articulated leg, we first evaluated different implementation environments. In the tutorials RBDL [7] and C++ was used. RBDL, the Rigid Body Dynamics Library, is an open-source software developed by Martin L. Felis from the university of Heidelberg. The main features of the library are the efficient computation of a robot's Dynamics and Kinematics, like computation of Jacobians, contact constraints, and inverse Kinematics. Nevertheless, we had a look at different development environments. The main advantages and disadvantages of the different environments are described in detail in the following.

MATLAB with Simscape Multibody: Simscape [6] is an Add On for MATLAB and provides a multibody simulation environment. It is easy to use for prototyping because of its intuitive syntax. Components like joints and links can be added by drag and drop and can then be connected via lines. For that, it is straightforward to assemble complex systems. Moreover, it is possible to define constraints between bodies. In addition to that, Simscape already provides a 3D simulation that visualizes the system. But we decided not to use Simscape Multibody because it would have become difficult to access the Jacobians and compute the control law.

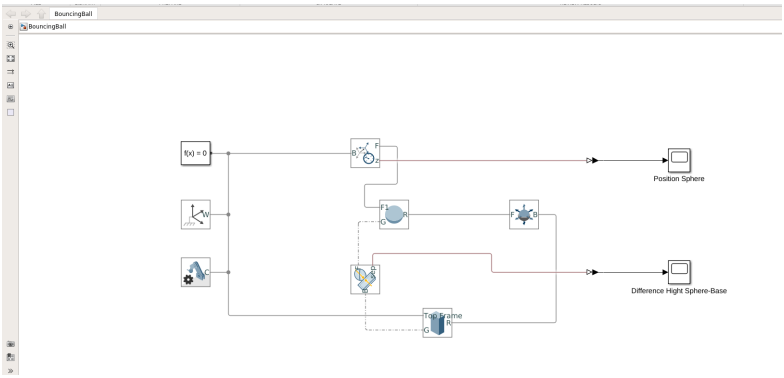

Fig. 3: Example for a Simscape Mode of a bouncing ball

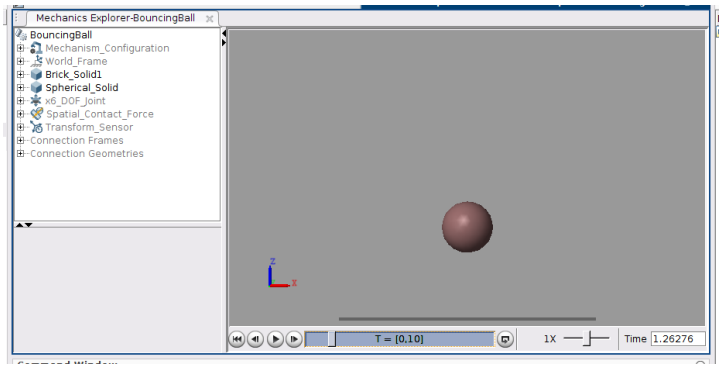

Fig. 4: Example for a Simscape Simulation of a bouncing ball

MATLAB with RBDL: MATLAB is a more convenient programming language for prototyping than C++. RBDL provides no wrapper for MATLAB, which is why we needed to create this ourselves. However, the attempt to implement the wrapper led to two problems: Firstly, the suboptimal integration of c++ libraries in MATLAB, which did not provide error messages. Secondly, the sheer number of methods and objects needed to wrap would take a long time.

MATLAB with Quindim: Quindim [4] is a robot dynamics library for MATLAB. The library is still a work in progress and therefore does not provide all functions that were needed for the implementation of the articulated leg. Further problems were the minimalistic documentation and that the library is only compatible with MATLAB versions up to 2017.

C++ with RBDL: With C++, all RBDL functions are available and easy to access. Furthermore, RBDL provides detailed documentation of its functions. On the one hand, C++ is a high-performance programming language, and we already had examples from the tutorials. On the other hand, it is not convenient for fast prototyping. For that reason, we decided not to use RBDL with C++.

Python with RBDL: RBDL already provides a wrapper for python. To verify that it works, we implemented the double pendulum and the compass gait walker from the tutorials. By doing so we realized that some functions that we would need for the implementation of the articulated leg were missing (`ComputeConstraintImpulsesDirect` and `CalcConstraintJacobian`) and some were wrongly wrapped

(`CalcBodySpatialJacobian`). Section III-A describes how we adjusted the wrapper accordingly to fix those missing functions. All in all, python is very convenient for prototyping, and it is easy to implement system visualizations. Therefore, we decided to implement the articulated leg with python and RBDL.

A. RBDL python wrapper modification

This section should give an overview, how we achieved adding missing functions to the python wrapper. (All functions which are already wrapped can be found in `rbdl-wrapper.pyx`). This wrapper is built using cython, a programming language that allows combining C code with python code to generate a python library. Before modifying the wrapper, it is essential to install all dependencies required for the build process:

```
sudo apt-get install python-dev build-essential
sudo pip3 install Cython
sudo pip3 install scipy
```

As an example, we explain the process of wrapping the function `ComputeConstraintImpulsesDirect`. For this, we need to modify the `crbdl.pxd` and the `rbdl-wrapper.pyx` file. Former is similar to a C header file and therefore tells which functions and which classes from the C library should be callable by cython. This file is extended with the declaration of the function. The declaration and in which class the function is defined can be found in the official RBDL documentation [5]. In the case of `ComputeConstraintImpulsesDirect` we add the following code to the section for `Constraints.h`:

```
cdef void ComputeConstraintImpulsesDirect (
      Model &model,
      const VectorNd &Q,
      const VectorNd &QDotMinus,
      ConstraintSet &CS,
      VectorNd &QDotPlusOutput)
```

The second step is to define the python equivalent of this function in `rbdl-wrapper.pyx` and how the datatypes between those functions have to be converted. Since there are already many different functions wrapped, this file provides many examples how to convert nontrivial datatypes and how to use the helper functions (`NumpyToVectorNd()`, `VectorNd.fromPythonArray()`, ...). Classes like `ConstraintSet` and `Model` save their C representation in the `thisptr[0]` property. In the case of `ComputeConstraintImpulsesDirect` one of the input values (`qdot_plus`) is passed as a reference and therefore needs to be modified. Because of this, we need to copy the entries of the wrapped array back to the python array:

```
def ComputeConstraintImpulsesDirect(Model model,
np.ndarray[double, ndim=1, mode="c"] q not None,
np.ndarray[double, ndim=1, mode="c"] qdot_minus not None,
ConstraintSet cs,
np.ndarray[double, ndim=1, mode="c"] qdot_plus not None
):
   qdot_plus_wrap = VectorNd.fromPythonArray(qdot_plus)
   crbdl.ComputeConstraintImpulsesDirect(
      model.thisptr[0],
      NumpyToVectorNd(q),
      NumpyToVectorNd(qdot_minus),
      cs.thisptr[0],
      (<VectorNd>qdot_plus_wrap).thisptr[0]
   )
   for i in range(len(qdot_plus)):
      qdot_plus[i] = qdot_plus_wrap[i]
```

When those changes are done, the library is build by executing the following commands in the rbdl-orb folder (or execute dobuild.sh):

```
sudo rm -rf build
mkdir build
cd build
cmake ..
ccmake ..s
sudo make install
```

IV. Implementation

A. Visualization

Even though MeshUp [3], which is compatible with RBDL, already provides some required visualization functionalities, it has some downsides: First, it is not possible to view states while running the code, which means that in the replay one must guess which line of code was responsible for which movement. Furthermore, visualizing custom objects (e.g., the SLIP-Model in addition to the articulated leg) is not achieved easily. This was the motivation for creating our visualization tool using matplotlib called `RobotPlotter`.

1) `RobotPlotter(model, stopFunction = None):` Initializes RobotPlotter with `model` as RBDL-model to be visualized. It is possible to pass an `stopFunction`, which is called when the "STOP"-Button within the UI is clicked. This function could (for example) stop the integration.

2) `RobotPlotter.addBodyId(id):` This functions needs to be called for each body which should be visualized since there is no easy way to get all bodies from RBDL. Example:

```
for id in ["floatingBase","link1","link2", "foot"]:
      robotPlotter.addBodyId(self.model.GetBodyId(id))
```

3) `RobotPlotter.setAxisLimit((-xlim, +xlim), (-ylim, -ylim)):` Sets the axis limits for the matplotlib visualization.

4) `RobotPlotter.print(t,text):` Prints the `text` to the console. But similar to all following functions, it also stores that this `text` has been printed at a given time `t`. This is used to print the text again when the playback-mode (Sec. IV-A8) is activated.

5) `RobotPlotter.updateRobot(t, q):` Updates the visualized robot with the provided q-values (as `np.array(q_size)`).

6) `RobotPlotter.showVector(t, origin, vector, color='r'):` This function visualizes a `vector` (displayed as an arrow) at a given `origin` with a certain `color` ('r' for red, 'b' for blue, ...). The function always displays just the most recent vector with the same color - which means that all previously added vectors with the same color are hidden. Therefore for visualizing more than one vector at once, different colors must be chosen.

7) `RobotPlotter.showPoints(t, style, point1, point2, ...):` This function can be used to display custom points and lines. How these points are visualized is determined by the `style` [2] property. E.g. "rx" are red points marked with an x. "ko-" is a black line spanned between all points with dots as point indicator. Similar as in `showVector()` (Sec. IV-A6), only the most recent version with the same style is displayed.

8) `RobotPlotter.playbackMode():` It should be called, when no new states have to be calculated, and therefore the UI-controls for activating the playback should be displayed.

B. Lua Model

In RBDL, it is possible to use the LuaModel Add-on. As described in the `README.md` `BUILD_ADDON_LUAMODEL` has therefore be set to true in the cmake file. The Lua Model is used to define the robot's design and can then be loaded into the code. The Lua Model of the articulated leg is described in the `articulated_leg.lua` file. First, all the body properties like length, mass, and inertia tensors of the links and the radius of foot and base are defined.

```
-- Body properties
length_link1 = 0.2
m_link1 = 0.1
I_link1_xx = (1/12)*m_link1*width_link1*width_link1
[...]
-- Matrix definitions
inertiaMatrix_link1 = {
   {I_link1_xx ,0.   ,0.  },
   {0.  ,I_link1_xx  ,0.  },
   {0.  ,0.       ,I_link1_zz}
}
[...]
```

Then the degrees of freedom of the joints are defined via matrices. The first three entries are for the rotational degrees of freedom, the last three for the directional ones. The floating base can move freely in x- and y-direction. All other joints can only rotate around the z-axis.

```
-- Making a table of joints
joints = {
   rotational_z = {
      {0.,0.,1.,0.,0.,0.}
```

```
    },
    floating = {
       { 0., 0., 0., 1., 0., 0.},
       { 0., 0., 0., 0., 1., 0.}
    }
}
```

After that the visualization properties like dimension and color of the bodies are defined.

```
-- Making the meshes
meshes = {
    link1 = {
       dimensions = { width_link1, length_link1, width_link1 },
       color = { 1, 0, 0 },
       mesh_center = { 0, -length_link1/2, 0 },
       src = "unit_cube.obj",
    },
    [...]
}
```

In the end, the model is defined. Each body gets a name as an individual identifier and a parent, i.e. the previous body it is connected to. Because the articulated leg is a walking robot that is not fixed to one point in space, the parent of the floating base is described as `ROOT`. The properties are assigned to the body and the joint. The `joint_frame` defines the point at which the body should get attached to its parent. Finally, the model gets returned.

```
-- Making the model
model = {
    frames = {
       {  name  = "floatingBase",
          parent = "ROOT",
          visuals = {meshes.floatingBase},
          body  = bodies.floatingBase,
          joint = joints.floating,
          joint_frame = {
             r = {0, 0, 0},
          }  },
       [...]
    }
}
return model
```

In the python code the Lua Model then gets loaded via `self.model = rbdl.loadModel(<path to the .lua file>)`. The bodies with their properties can be accessed via `self.model.GetBodyId(<name of the body>)`. In addition to that information like the number of degrees of freedom can be read from the model.

C. Stance Controller

During stance phase the stance controller is active and `controllerStance(state, t)` gets called. The goal of the operational space controller is to calculate the torques of the hip and knee joint such that the leg follows the motion of the SLIP model. First, all the matrices needed for calculation get computed. The Jacobian of the center of gravity is calculated via

$$\boldsymbol{J}_{cog} = \sum_{n=1}^{\#bodies} \frac{J_n m_n}{\sum_{i=1}^{\#bodies} m_i} \tag{9}$$

To compute the Coriolis and centrifugal terms, the RBDL function `NonlinearEffects(model, q, qDot, tauOut)` is used. The output is defined by the torque the leg needs to compensate for the effects of gravitational, Coriolis, and centrifugal forces.

```
    def controllerStance(self, state, t):
        [...]
        # b coriolis, centrifugal, gravitational terms
        b = np.zeros(self.model.q_size)
        rbdl.NonlinearEffects(self.model, state.q, state.qd, b)
```

In M. Hutters paper SLIP Running with an Articulated Robotic Leg [8], he describes the task inertia as

$$\boldsymbol{\Lambda}^* = (JM^{-1}SN_sJ^*)^{-1} \tag{10}$$

Performing a matrix multiplication on M of dimension 4x4 and N_s of dimension 2x4 leads to an dimension error. With the help of the other project group we used the term

$$\mathbf{\Lambda}^* = (JM^{-1}(SN_s)^T J^{*T})^{-1} \tag{11}$$

derived by L. Sentis in his dissertation about synthesis and control of whole-body behaviors in humanoid systems [9].

```
        [...]
        lambda_star = np.linalg.inv(J_cog[0:2] @ M_inv @ (S @ N_s).T @ J_star.T)
        # problem with dimensions
        # lambda_star = np.linalg.inv(J_s @ M_inv @ S @ N_s @ J_s.T)
```

To calculate the difference between initial and current SLIP length $deltaX$, needed for the calculation of F_{leg}, we computed the current SLIP length by determining the foot's position and the CoG of the leg and subtracted it with the initial SLIP length $self.slip\_length$. In the end, all terms get assembled to compute the resulting torque.

```
        [...]
        # compute distance foot and COM
        distance_foot_com = com - footpos
        slip_new_length = np.linalg.norm(distance_foot_com, ord=2)
        angle = np.arctan2(distance_foot_com[1],distance_foot_com[0])
        deltaX = self.slip_length - slip_new_length
        [...]
        slip_force = self.slip_stiffness*(np.array([deltaX*math.cos(angle),
            deltaX*np.sin(angle), 0])) + self.slip_mass*gravity
        [...]
        # Control law
        torque_new = J_star.T @ (lambda_star @ ((1/self.slip_mass)*slip_force))
        coriolis = J_star.T @ coriolis_grav_part
        [...]
        state.tau[2:4] = coriolis + torque_new
        return state
```

D. Flight Phase Controllers

During the flight phase, a PID controller is active, which adjusts the leg in operational space. This flight phase is divided into three regions, with each one having different target values:

```
def controllerFlight(state, t):
   [...]
   #phase 1
   [...]
   #phase 2
   [...]
   #phase 3
   [...]
   #calcuate tau from pid
   tau = self.pid(state.q[2:4])
   state.tau[2:4] = tau

   self.flight_last_height = com[1]
```

The first phase is activated instantly at the beginning of the flight phase until a given height. During this phase, the PID controller tries to keep the angles of the leg constant. This is important since pulling the leg forward in this position would lead to the leg touching the ground again immediately.

```
   foot = rbdl.CalcBodyToBaseCoordinates(self.model, state.q, self.model.GetBodyId("foot"),
       np.zeros(3), False)
   [...]
   #phase 1
   if self.flight_target_q is None and foot[1] < 0.3:
      #started -> keep let position until foot is high enough and could touch ground if moving
      self.pid.setpoint=state.q[2:4]
   [...]
```

When the leg jumped high enough, the next phase becomes active until the apex is reached. Here the PID controller adjusts the leg to some initial value so that only minor changes are required when the actually desired impact angle is calculated.

```
    [...]
    #phase 2
    if self.flight_target_q is None and foot[1] >= 0.3:
        #move leg to initial position
        self.pid.setpoint=self.impactQ[0:2]
    [...]
```

When the leg reaches the apex, we calculate the desired angle for the impact. For this, the height of the apex is compared with the last apex height. The difference between these values defines how the impact angle should defer from the initially defined angle. This angle and the SLIP length is then used to calculate the desired operational space coordinates using inverse kinematics.

```
    [...]
    #phase 3
    if self.flight_last_height > com[1] and self.flight_target_q is None:#is apex?. 3rd phase
        #apexP controller
        angle_des = self.impactAngle - 0.45 * (self.apex_des - com[1])
        new_point = np.array([
            com[0]+math.cos(angle_des)*self.slip_length_zero,
            com[1]-math.sin(angle_des)*self.slip_length_zero,
            0])
        self.apex_des = com[1]

        #inverse kinematics
        [...]
        basePos = np.array([state.q[0], state.q[1], 0]#floating base should keep position
        target_coord = np.array([basePos, np.array([new_point[0],new_point[1],0])])#food should
            move to this point
        target_q = np.zeros(self.model.q_size)
        rbdl.InverseKinematics(
            self.model,
            state.q,
            np.array([self.model.GetBodyId('floatingBase'), self.model.GetBodyId('foot')]),
            np.zeros([2,3]),
            target_coord,
            target_q)
        self.flight_target_q = target_q
        #set target
        self.pid.setpoint=self.flight_target_q[2:4]
    [...]
```

E. Impact Compensation

As described in the theory part impact compensation gets introduced to achieve continuous hopping without energy loss. Therefore a new length of the SLIP model gets calculated. This happens in the `solve()` function when the leg switches from flight to stance phase. Before the velocity change due to impact collision between foot and floor happens, `calculate_new_slip_length(state, t)` is called. The function first calculates all helper variables needed to calculate the energy difference and the corresponding new SLIP length. `state.qd` describes the pre-impact velocity of the articulated leg. In the last step the new Δl, here called `new_length`, gets added to the initial SLIP length. Every time the articulated leg switches from flight to stance mode, the process is repeated.

```
    def calculate_new_slip_length(self, state, t):
        # calculate Jacobians J_cog, J_s, mass matrix, space inertia lambda_s and null space N_s
        [...]
        # helper var
        matrix = J_cog.T @ J_cog - N_s.T @ J_cog.T @ J_cog @ N_s
        # calculate new length
        new_length = (1/self.slip_stiffness) * self.slip_mass * (state.qd.T @ matrix @ state.qd)
        new_length = math.sqrt(abs(new_length))
        # set new slip length
        self.slip_length = self.slip_length_zero + new_length
```

F. g Function

As stated in chapter II the SLIP model alters between a stance and a flight phase. Therefore, the `g_function` is responsible for detecting the event for the two phases and activating the respective controller. For the stance phase, the exit condition depends on the spring force. As long as the stance phase is currently active, the current spring force is calculated. As long as the articulated leg is in an upwards motion, i.e., the spring force is decreasing, and as soon as the spring force falls below a defined limit, the flight controller is activated. Due to numerical inaccuracy, the limit is not exactly set to zero. In addition, the stance phase is left if the distance between the floating base and the foot exceeds a defined boundary. This prevents the articulated leg from fully stretching.

```
    def g(t, y):
        [...]
        state = State(self.model, y)

        if self.mode == Mode.STANCE:
              # calculate slip force
              distance_foot_com = com - footpos
              slip_new_length = np.linalg.norm(distance_foot_com, ord=2)
              f = self.slip_stiffness*(self.slip_length - slip_new_length) +
                  self.slip_mass*self.model.gravity[1]
              [...]
              #only detect if spring force decreases
              if self.lastSpringForce > f:
                 floatingBasePos = rbdl.CalcBodyToBaseCoordinates(self.model, state.q,
                     self.model.GetBodyId("floatingBase"), np.zeros(3), False)
                 baseFootDist = np.linalg.norm(floatingBasePos-footpos, ord=2)
                 if f < 0.08 or abs(baseFootDist) > 0.75:#f small or fully extended
                    self.compute_inverse_kinematics = True
                    self.lastSpringForce = -10000
                    return 0#STOP
    [...]
    g.terminal = True #stop integration when g function raises event
    [...]
    #toggle phase
    self.mode = Mode.FLIGHT if self.mode == Mode.STANCE else Mode.STANCE
```

If the `g_function` raises an event, the integration is stopped, and the activated phase, i.e., the flight phase, is saved in the variable `self.mode`. The height of the foot position defines the exit condition of the flight phase. If the articulated leg is in a downward motion and as soon as the foot touches the ground, the flight phase terminates, and the stance phase controller is activated by setting `self.mode == Mode.STANCE`.

```
    def g(t, y):
        [...]
        state = State(self.model, y)
              [...]
        if self.mode == Mode.FLIGHT:
                         [...]
              if self.lastFootPos > footpos[1] and self.lastFootPos > 0:
                 if footpos[1] <= 0:
                    self.lastFootPos = -10000
                    return 0 #STOP
                 else:
                    return footpos[1]
    [...]
    g.terminal = True #stop integration when g function raises event
        [...]
    #toggle phase
        self.mode = Mode.FLIGHT if self.mode == Mode.STANCE else Mode.STANCE
    #compute impulses
    if self.mode == Mode.STANCE:
        self.calculate_new_slip_length(state, t)
        rbdl.ComputeConstraintImpulsesDirect(self.model, x[:self.dof_count], x[self.dof_count:],
            self.constraintSet(), qd_plus)
        x[self.dof_count:] = qd_plus
```

After the stance phase is entered, the impulses are calculated based on the new SLIP length from the impact compensation, as mentioned in the chapter before.

V. Results

With the implemented code structure, a stable motion of the articulated leg could be achieved. To illustrate the effect of the impact compensation, the simulation results of the implementation with and without impact compensation are compared. In the following figure 5 you can see our simulation results without Impact compensation visualized with the customized feature presented in chapter IV.

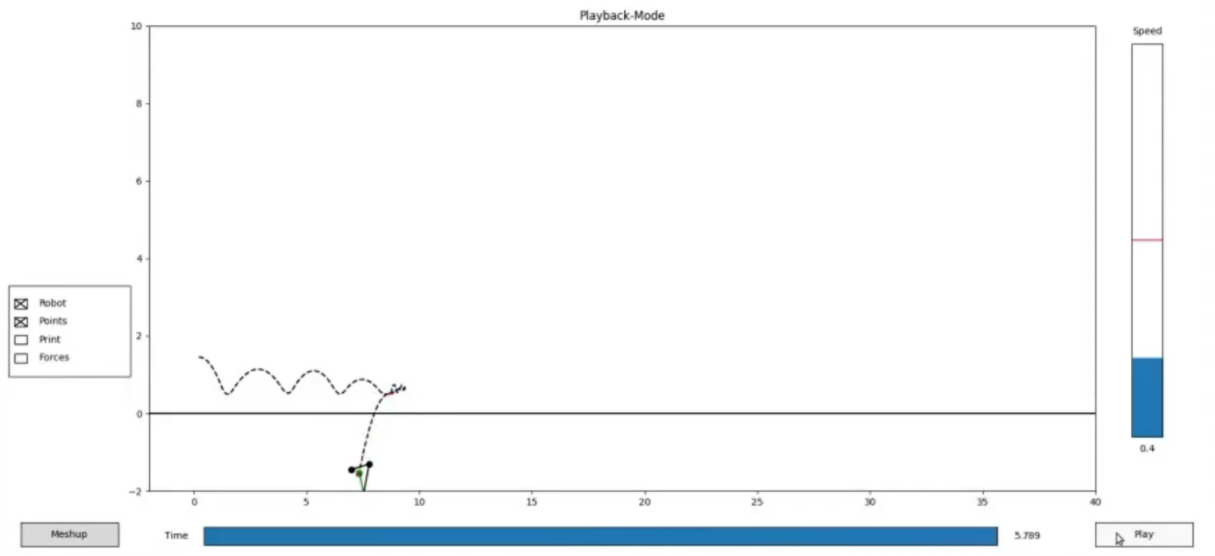

Fig. 5: Simulation results without impact compensation

The SLIP model is displayed in green, whose dynamic is used for the articulated leg. The black lines are the two segments of the articulated leg, and the dotted line describes the motion of the Center of Mass of the articulated leg. Without the impact compensation, the leg falls over after five steps.

In contrast to that, the implementation with impact compensation achieves a stable motion. The COM always reaches approximately the same apex height instead of decreasing the apex height due to the lost energy at collision as illustrated in figure 6.

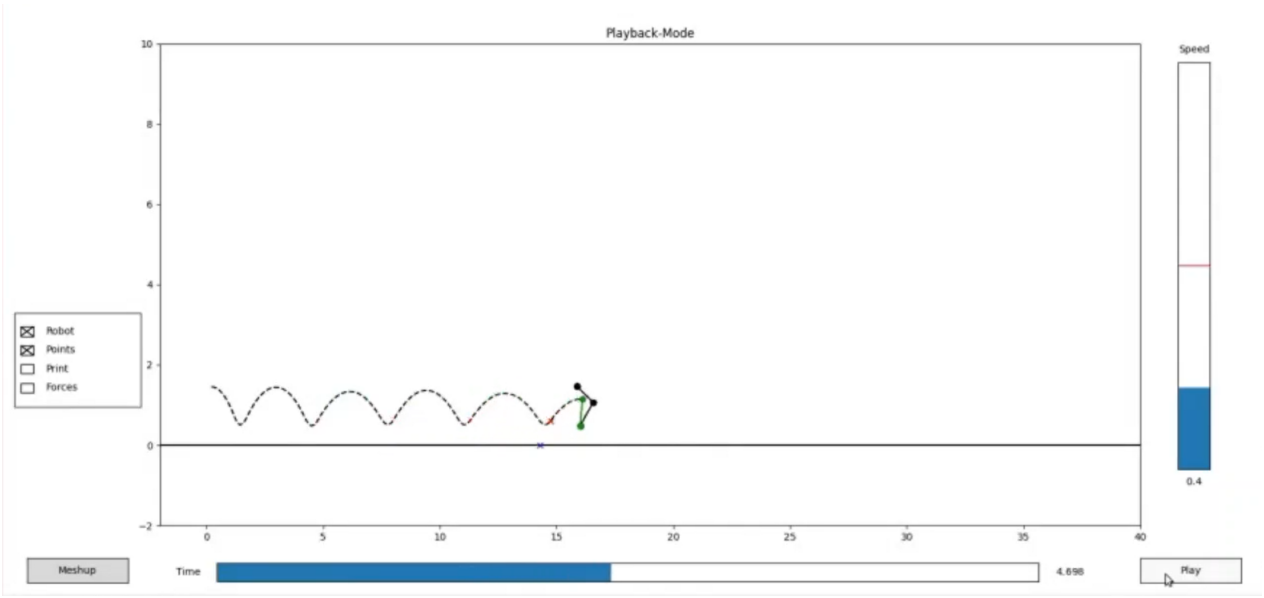

Fig. 6: Simulation results with impact compensation

Another point to mention is that over the course of parameter tuning a conspicuous behaviour was observed, that so far could not have been resolved. For significantly decreasing the mass of the SLIP model as well as the articulated leg, the behaviour shows a gain in the kinetic energy during each stance phase. The expected behaviour would be an approximately constant

level of kinetic energy at each entering point of the stance phase. For this reason the mass parameters were chosen at a level, where this effect appeared to have a negligible influence. Note that the code structure with the current parameters is provided in our GitLab repository [1].

VI. Outlook

All in all, the articulated leg based on the SLIP model presented in the paper of Marco Hutter [8] was implemented successfully. Also, the realized approach allows for fast testing of model parameters due to its customized visualization. In the course of the project, a python wrapper for the RBDL functions has been enhanced for fast and prototype-friendly code generation. An extension for future work would be to test the chosen parameter-set on a real robot and, if needed, further optimize parameterization.

References

[1] Gitlab repository dodo gruppe 1. https://gitlab.lrz.de/00000000014A937A/dodo-gruppe-1.git.
[2] matplotlib style property. https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html. (Accessed on 20/03/2021).
[3] Meshup github. https://github.com/ORB-HD/MeshUp. (Accessed on 20/03/2021).
[4] Quindim github. https://github.com/brunolnetto/quindim. (Accessed on 01/03/2021).
[5] Rbdl documentation. https://rbdl.github.io/index.html. (Accessed on 28/03/2021).
[6] Simscape multibody. https://uk.mathworks.com/products/simmechanics.html?s_tid=srchtitle. (Accessed on 01/03/2021).
[7] Martin L. Felis. Rbdl: an efficient rigid-body dynamics library using recursive algorithms. *Autonomous Robots*, pages 1–17, 2016.
[8] Marco Hutter, C. Remy, Mark Hoepflinger, and Roland Siegwart. Slip running with an articulated robotic leg. pages 4934–4939, 10 2010.
[9] O. Khatib and L. Sentis. Synthesis and control of whole-body behaviors in humanoid systems. 2007.
[10] Andre Seyfarth, Hartmut Geyer, Michael Günther, and Reinhard Blickhan. A movement criterion for running. *Journal of Biomechanics*, 35(5):649–655, 2002.