Final Report Whole-Body Balance Control

Katharina Hermann, Sadek Cheaib
Supervision: Dennis Ossadnik

I. Introduction

A. Motivation and Related Work

The overall goal of robots is to move or to balance in challenging environments by avoid tipping over and falling, especially against external disturbances (for example in human robot interaction). To realize this, we need a good control strategy, which keeps the dynamic system state of the robot inside a defined state space region by applying a suitable wrench at the COM (Centroidal dynamics) to compensate gravity and to keep dynamic balance adapting (compliantly) to unknown external perturbation forces. Since the robot has a floating-based structure, we can not apply the wrench at the COM directly, but we have to generate it via external forces at the contact points by taking into account the dynamics of the whole floating-body. The goal of our work, was to implement a balancing control algorithm for Dodo, a 12 DOF biped robot using the simulation framework Mujoco.

There are 3 fundamental different approaches to realize the goal of balancing.

1) **Joint Position-based compliance control:**
Dynamics based walking pattern generation provides desired trajectories for underlying position controllers. Due to the finite foot support area pure position control is insufficient for executing these trajectories. Therefore, one integrates a inner force/ ZMP (Zero Moment Point) control loop via force sensing at every contact point. As it is computationally very inefficient it leads to a bad real-time behavior with time delays in the controller [1]. Therefore, this approach allows a generation of a large range of stepping and walking motions but is restricted to a limited set of contact states.
2) **Joint momentum-based control:**
A joint momentum-based balancing controller is realized by controlling the desired ground reaction force and center of pressure at each support foot independently [2]. The control strategy focuses on determining and compensating linear and angular momentum with inverse dynamics from desired forces at feet. Exact inverse dynamics calculation is problematic as it requires an exact model including the inertia matrix, which becomes impractical for large-DoF systems.
3) **Joint torque-based control:** Our work chooses a joint torque-based control approach. We build on the work of [3], [4], and [5]. All of these works use a torque-based control approach for robot balancing against unknown disturbances. Works before have mostly only studied this approach for biped walkers [6] and robot grasping [7]. In force control, the input for the controller is directly a desired torque in each joint, which can then for example be realized via series elastic actuation.

B. Problem Statement

Our approach is composed of 3 main steps, summarized in Figure 1. We will explain each step with in the following. Additionally we present our implementation with the Mujoco framework for each step.

1) Set a desired wrench at the COM F_{COM} from the robot to the environment.
2) Distribute F_{COM} among predefined contact points: F_C
3) Transform F_C to the joint torques τ directly via the Jacobian matrix J_{Ci}.

C. Main advantage of the approach

The required wrench F_{COM} at the COM is distributed among predefined contact points via a constrained optimization problem. This approach ensures a positive contact forces via a contact force constraint. Further, for the distribution of F_{COM} the paper reuses a formulation from grasping. With this approach the authors realize a controller,which keeps both the position and orientation of the robot.

With the paper's torque force-based approach, there is no need for contact force measurement as for joint position-based compliance control and for a calculation of inverse kinematics or dynamics. Instead, the torques are obtained from the contact forces with a passivity based compliance approach.

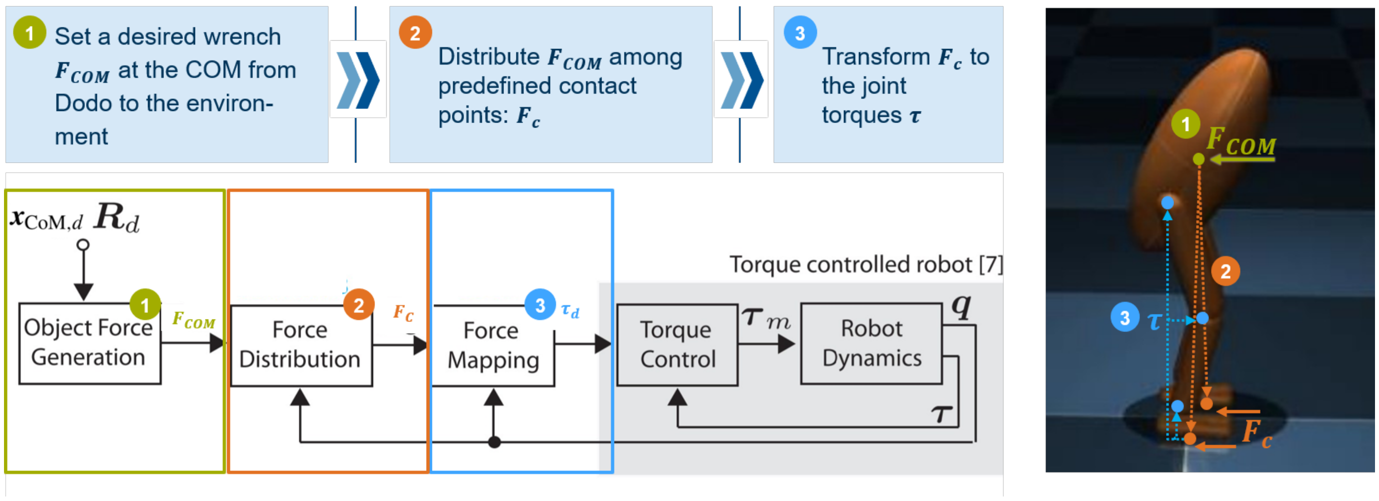

1 Set a desired wrench F_{COM} at the COM from Dodo to the environment
2 Distribute F_{COM} among predefined contact points: F_c
3 Transform F_c to the joint torques τ
$x_{\mathrm{CoM},d}$ R_d
Object Force Generation
F_{COM}
Force Distribution
F_c
Force Mapping
τ_d
Torque controlled robot [7]
Torque Control
τ_m
Robot Dynamics
q
τ
F_{COM}
τ
F_c

Fig. 1: Overview of the balancing controller

II. Methodological approach

A. COM Wrench Generation F^d_{COM}

The dynamical equations, which the whole approach is based on, are described in Eq. (1) in the world frame W. The first 6 equations describe the dynamics of the COM with $x \in R^3$ being the position of the COM and $\omega \in R^3$ being the angular orientation of the COM. The last n equations describe the dynamics of the robot's controllable joints with $q \in R^n$ being the joint position parameter. As we formulated, the velocity w.r.t the COM, the first 6 and last n (number of DOF - here: 12) equations are only coupled via the wrenches at the contact points (the legs).

With this decoupled description we can directly control the desired position of the COM, with the joint torques by calculating the desired wrench at the COM F^d_{COM} (Fig.1, Step 1), mapping it via the the contact map G to the wrenches at the contact points F_c (Fig.1, Step 2), and then mapping it back via the Jacobian J to the control variables, the joint torques τ. (Fig.1, Step 3).

$$\begin{bmatrix} mI & 0 & 0 \\ 0 & M_{11} & M_{12} \\ 0 & M_{21} & M_{22} \end{bmatrix} \begin{pmatrix} \ddot{x}_{COM} \\ \ddot{\omega}_{COM} \\ \ddot{q} \end{pmatrix} + \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} \begin{pmatrix} \dot{x}_{COM} \\ \dot{\omega}_{COM} \\ \dot{q} \end{pmatrix} + \begin{pmatrix} mg_0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \tau \end{pmatrix} + \begin{bmatrix} G \\ J^T \end{bmatrix} F_c \tag{1}$$

For the 1st step the objective is to bring the wrench at the COM to a desired value F^d_{COM}. This represents the desired task of balancing the COM while guaranteeing the gravity compensation. The respective force components f^d_{COM} (Eq. 2) are computed with a PD-like posture controller, to control the position of the COM. f^d_{COM} is the desired force for recovering the initial position, which can be computed according to Eq. (3) from a PD feedback law with K_t the proportional and D_t the differential gain matrix.

$$F^d_{COM} = \begin{pmatrix} f^d_{COM} \\ m^d_{COM} \end{pmatrix} \in R^{6\times 1} \tag{2}$$

$$f^d_{COM} = mg + K_t(x_{COM,d} - x_{COM}) - D_t\dot{x}_{COM} \tag{3}$$

The torque components ${}^{d}_{COM}$ for controlling orientation perturbations are computed with PD-like orientation controller. The torque m_r, which has to be applied to recover the initial orientation, is obtained according to (4) from a description of a torsional spring, which acts to align the desired and the current orientation. In (5), this torque m_r is then added to the term $D_r(\omega - \omega_d)$ to additionally control the angular velocity.

$$m_r = -2(\delta I + \hat{\epsilon})K_r\epsilon \tag{4}$$

$$m^d_{COM} = R_b^T(m_r + D_r(\omega - \omega^d)) \tag{5}$$

B. Force distribution F_c

After computing the desired wrench at the COM F^d_{COM}, we now want to distribute it to the contact points, to obatin the forces F_C acting at the contact frames. For balancing one can use a similar approach to grasping, where the forces at several contact points generate a net wrench at an object. In this case, the contact forces f_i get mapped to the net wrench via the grasp map G, according to (6). As shown in (7), G is composed of B_i, the wrench basis (characterizing the contact model - here one frictional contact point - and mapping the force f_i to a wrench) and W_P, the wrench transformation matrix, which is the transposed adjoint transformation matrix Ad^T_{OP}, to map forces from the contact frame P to the origin O.

$$F_O = \sum\nolimits_i^{\eta} G_i f_i = G f_C \tag{6}$$

$$G_i = W_{Pi} B_i = Ad^T_{OP} B_i = \begin{bmatrix} R_P & 0 \\ \hat{r}_P & R_P \end{bmatrix} \begin{bmatrix} I \\ 0 \end{bmatrix} \tag{7}$$

A biped robot can also be analyzed as a series of contact forces applied at the contact points, which generate a net wrench on the robot as here on the object for the grasping case. For balancing however, the problem is inverted: One first computes desired wrench at the COM of the robot, and applies it then to the contact point forces, instead of reacting to them. The distribution of the desired wrench is again given by (8). Instead of only computing only the contact forces f_C, we compute the whole contact wrench F_c. G is now called the contact map and is computed according to (9) with the transposed adjoint matrices $Ad^T_{i_c}$ ($i \in [1, m]$ and m, the number of contact points) and B_i, the wrench basis, being the identity I. $\hat{x}_{i_c}$ denotes the configuration-dependent lever arm between the end effector frame and the COM expressed in the world frame W.

$$F_{COM} = G F_c \tag{8}$$

$$G = \begin{bmatrix} Ad^T_{1_c} & ... & Ad^T_{m_c} \end{bmatrix} = \begin{bmatrix} I & 0 & ... & I & 0 \\ \hat{x}_{1_c} & I & ... & \hat{x}_{m_c} & I \end{bmatrix} \in R^{6\times 6m} \tag{9}$$

In order to solve for F_C, we first have chosen the simple approach of computing the Pseudo Inverse of the contact map $G^\dagger = G^T(GG^T)^{-1}$. We then can compute the vector for the m contact wrenches F_C , simply by Eq. (10).

$$F_C = G^\dagger F_{COM} \in R^{6m} \tag{10}$$

However, this approach does not consider the optimal distribution of the contact and we do not respect constraints such as a positive z-component f_{i_z} or an optimal friction cone of the x- and y-components f_{i_x} and f_{i_y} with μ being the friction coefficient (eq. 12). Therefore as an addition, we solve the constrained optimization approach introduced by [3]. With the weighted cost function (11) we can minimize the error for the contact point wrenches and the euclidean distance (magnitude) of the wrenches. To ensure positivity of contact forces (only pushing) and the fulfillment of the friction cone, the minimization is subjected to the inequality constraints (12).

$$\min_{F_C} (F_{COM} - GF_C)^T Q (F_{COM} - GF_C) + \alpha F_C^T F_C \tag{11}$$

subject to:

$$F_{Ci} = \{f_i \in R^3 | \sqrt{f^2_{i_x} + f^2_{i_y}} \le \mu f_{i_z}, f_{i_z} \ge 0\} \tag{12}$$

For the implementation in section V, we use a Quadratic programming framework, which uses linear constraints for computational efficiency. This corresponds to approximating our Coulomb firction cone constraint with an inner pyramidal friction cone constraint of the following form:

$$F_{Ci} = \{f_i \in R^3 || f_{i_x} + f_{i_y}| \le \mu f_{i_z}, f_{i_z} \ge 0\} \tag{13}$$

C. Force Mapping τ

In the third and last step, we want to perform the mapping from the contact wrench to joint torques, τ, which are then fed into the controller. Due to the decoupling of the equations, this can just be done via matrix multiplication of the transpose Jacobian $J^T \in R^{n\times 6}$ (from the contact frame w.r.t the COM expressed in the world frame W) with the contact wrenches F_C, as shown in (14). τ can now be fed in the controller of Fig. 1 as τ_d.

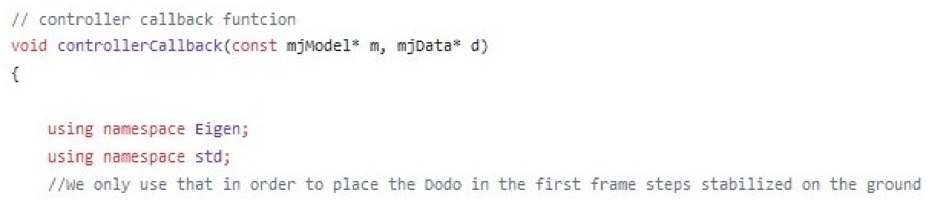
$$\tau = \sum\nolimits_{i=m} J_i^T F_{Ci} \tag{14}$$

III. Implementation

For the explanation of the implementation we will refer to the 3 main steps of Section II-A, II-B and II-C. The implementation can be found in our Git Repository under *this link*.

A. Initial Setup

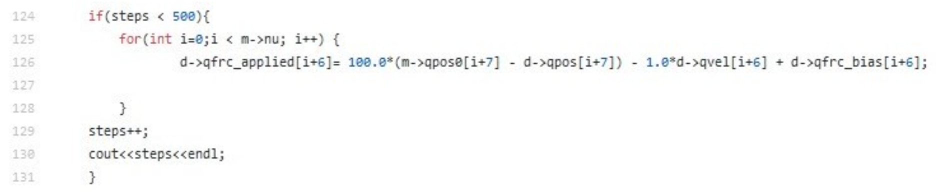
The whole algorithm is implemented in the *controllerCallback* function of Mujoco according to Fig. 2, which is called during the simulation main loop to set the control variables.



```
// controller callback funtcion
void controllerCallback(const mjModel* m, mjData* d)
{

    using namespace Eigen;
    using namespace std;
    //We only use that in order to place the Dodo in the first frame steps stabilized on the ground
```

Fig. 2: Controller Callback function for the implementation of the algorithm to calculate the control variables

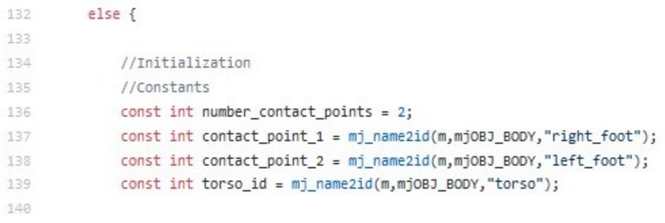
The robot is first stabilized with a normal PD controller according to Fig. 3 for the first time steps.



```
    if(steps < 500){
        for(int i=0;i < m->nu; i++) {
                d->qfrc_applied[i+6]= 100.0*(m->qpos0[i+7] - d->qpos[i+7]) - 1.0*d->qvel[i+6] + d->qfrc_bias[i+6];

        }
    steps++;
    cout<<steps<<endl;
    }
```

Fig. 3: Initial Stabilization of the robot in the first time steps of the simulation with a simple joint PD controller

After this time step, we start with the actual calculation of the control torques τ. First we initialize some general variables, according to Fig. 4. Then we initialize the variables (see Fig. 5), which we need for the equations of the COM Wrench Generation of Section II-A. Here we also do have some auxiliary variables. Especially important are the values for the gain and stiffness matrices K_t, D_t, K_r, D_r, which must be found empirically. Secondly, we initialize the variables for the Force Distribution according to Section II-B (see Fig. 6) and thirdly we initialize the variables for the Force Mapping equations of Section II-C (see Fig. 7).



```
    else {

        //Initialization
        //Constants
        const int number_contact_points = 2;
        const int contact_point_1 = mj_name2id(m,mjOBJ_BODY,"right_foot");
        const int contact_point_2 = mj_name2id(m,mjOBJ_BODY,"left_foot");
        const int torso_id = mj_name2id(m,mjOBJ_BODY,"torso");

```

Fig. 4: Overview of the balancing controller

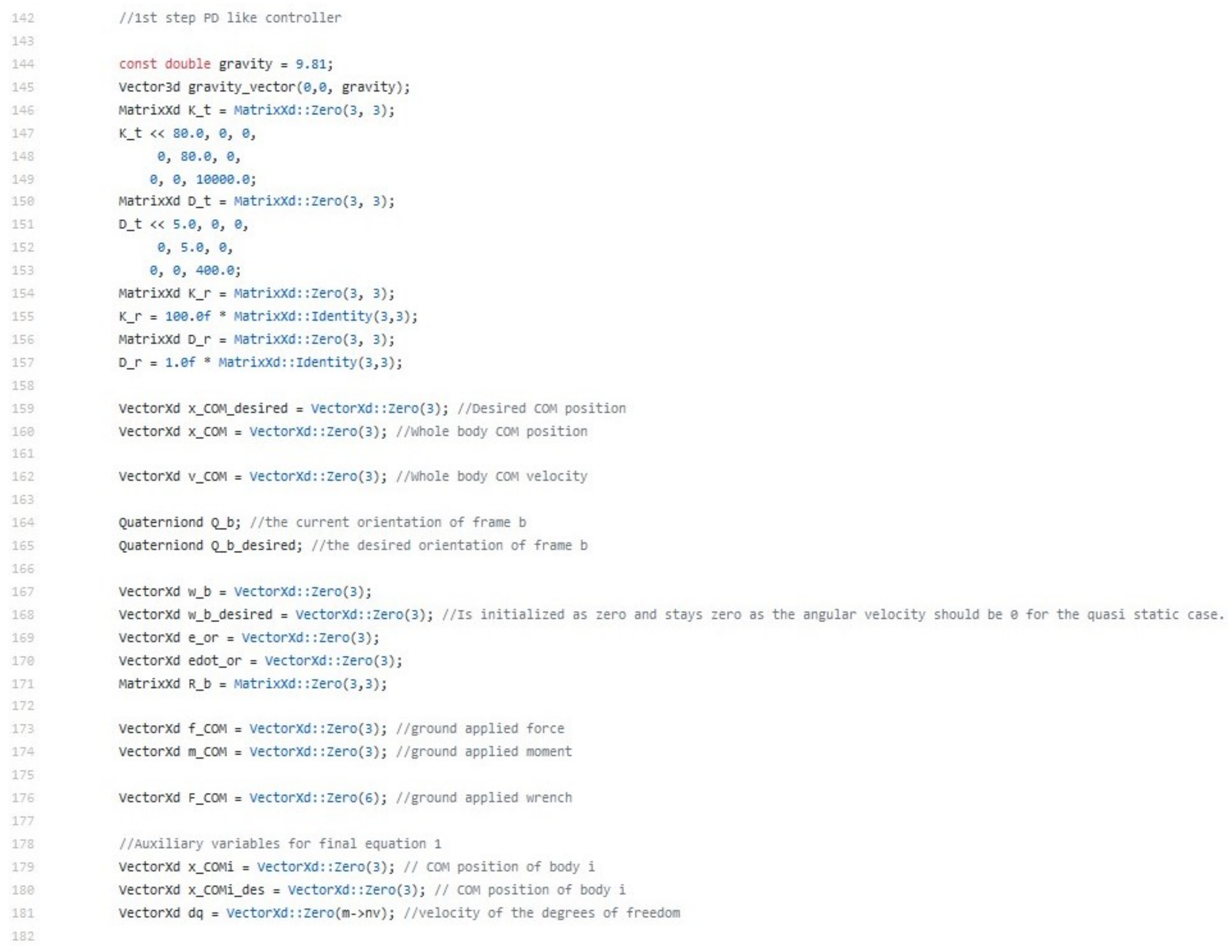
IV. Step 1 COM Wrench Generation

In a first step we calculate the variables for the PD like posture and position controllers. For the position controller (Eq. 3) we need the desired position of the COM $x_{COM,d}$ (*x_COM_desired*), and for the posture controller (Eq. 5), we need the quaternion representation of the desired base orientation ω^d (*Q_b_desired*). We calculate both according to Fig. 8, getting the initial values during the stabilization phase.

In Fig. 9 we first compute the variables for the position controller (Eq. 3) - the COM position x_{COM} (*x_COM*), and the COM Jacobian (*J_COM*) as an auxiliary variable in order to get the COM velocity $\dot{x}_{COM}$ (*v_COM*). We get both, the COM position and the Jacobian by computing the weighted sum of all individual body parts, with the mass of each sub-body (*m->body_mass[i]*) being the weight.

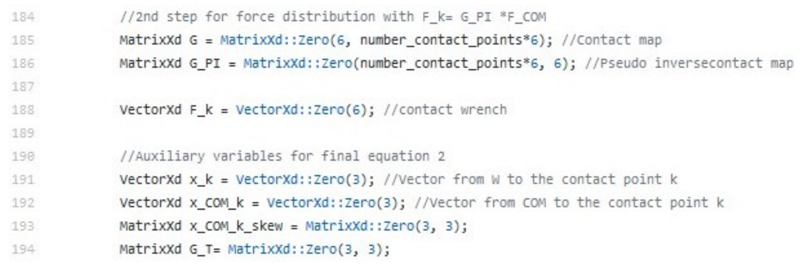
Afterwards we compute the variables for the posture controller (Eq. 5) - the current base orientation ϵ (*Q_b*) and the current angular velocity ω (*w_b*) of the base frame B. The base frame B is the frame of the torso of the robot and a reference frame for many following calculations.

Concluding Step 1 of Section II-A, those variables are used for the final calculation of the desired COM wrench (Eq. 2, 3, 5) according to Fig. 10 using the preimplemented function *virtualSpringPD* from the eigenUtils.cpp file.



```
//1st step PD like controller

const double gravity = 9.81;
Vector3d gravity_vector(0,0, gravity);
MatrixXd K_t = MatrixXd::Zero(3, 3);
K_t << 80.0, 0, 0,
      0, 80.0, 0,
     0, 0, 10000.0;
MatrixXd D_t = MatrixXd::Zero(3, 3);
D_t << 5.0, 0, 0,
      0, 5.0, 0,
     0, 0, 400.0;
MatrixXd K_r = MatrixXd::Zero(3, 3);
K_r = 100.0f * MatrixXd::Identity(3,3);
MatrixXd D_r = MatrixXd::Zero(3, 3);
D_r = 1.0f * MatrixXd::Identity(3,3);

VectorXd x_COM_desired = VectorXd::Zero(3); //Desired COM position
VectorXd x_COM = VectorXd::Zero(3); //Whole body COM position

VectorXd v_COM = VectorXd::Zero(3); //Whole body COM velocity

Quaterniond Q_b; //the current orientation of frame b
Quaterniond Q_b_desired; //the desired orientation of frame b

VectorXd w_b = VectorXd::Zero(3);
VectorXd w_b_desired = VectorXd::Zero(3); //Is initialized as zero and stays zero as the angular velocity should be 0 for the quasi static case.
VectorXd e_or = VectorXd::Zero(3);
VectorXd edot_or = VectorXd::Zero(3);
MatrixXd R_b = MatrixXd::Zero(3,3);

VectorXd f_COM = VectorXd::Zero(3); //ground applied force
VectorXd m_COM = VectorXd::Zero(3); //ground applied moment

VectorXd F_COM = VectorXd::Zero(6); //ground applied wrench

//Auxiliary variables for final equation 1
VectorXd x_COMi = VectorXd::Zero(3); // COM position of body i
VectorXd x_COMi_des = VectorXd::Zero(3); // COM position of body i
VectorXd dq = VectorXd::Zero(m->nv); //velocity of the degrees of freedom
```

Fig. 5: Initialization of the variables for Step 1



```
//2nd step for force distribution with F_k= G_PI *F_COM
MatrixXd G = MatrixXd::Zero(6, number_contact_points*6); //Contact map
MatrixXd G_PI = MatrixXd::Zero(number_contact_points*6, 6); //Pseudo inversecontact map

VectorXd F_k = VectorXd::Zero(6); //contact wrench

//Auxiliary variables for final equation 2
VectorXd x_k = VectorXd::Zero(3); //Vector from W to the contact point k
VectorXd x_COM_k = VectorXd::Zero(3); //Vector from COM to the contact point k
MatrixXd x_COM_k_skew = MatrixXd::Zero(3, 3);
MatrixXd G_T= MatrixXd::Zero(3, 3);
```

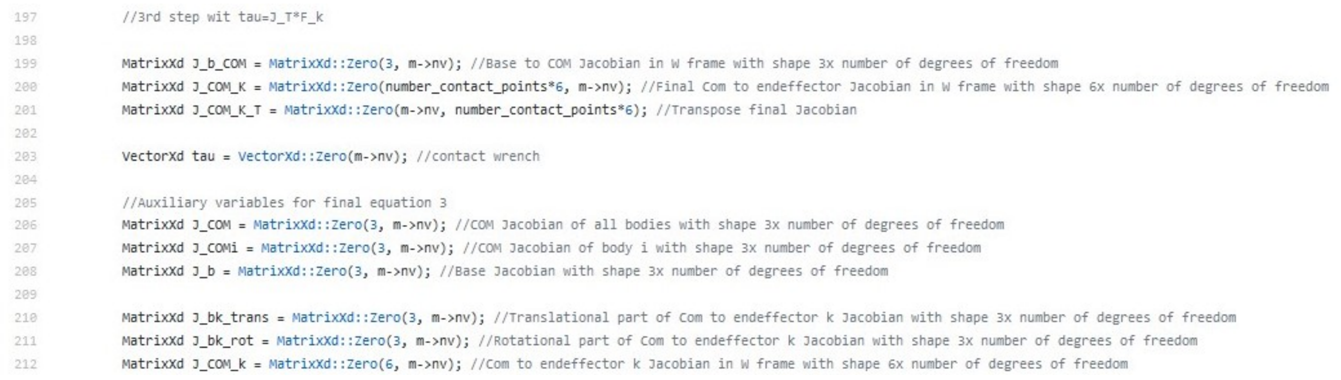
Fig. 6: Initialization of the variables for Step 2

V. Step 2 Force Distribution

In the Force distribution step according to section II-B we first get the positions of the 2 contact points (end effector legs) $\hat{x}_{ic}$ (x_COM_k) w.r.t the COM expressed in the world frame W according to Fig. 11. Inserting them in the matrix G leads to the contact map. The dimension of the contact map G is $R^{6\times 12}$ as we concatenate the contact wrenches of the 2 contact points in one vector $F_C \in R^{12\times 1}$ (F_k).

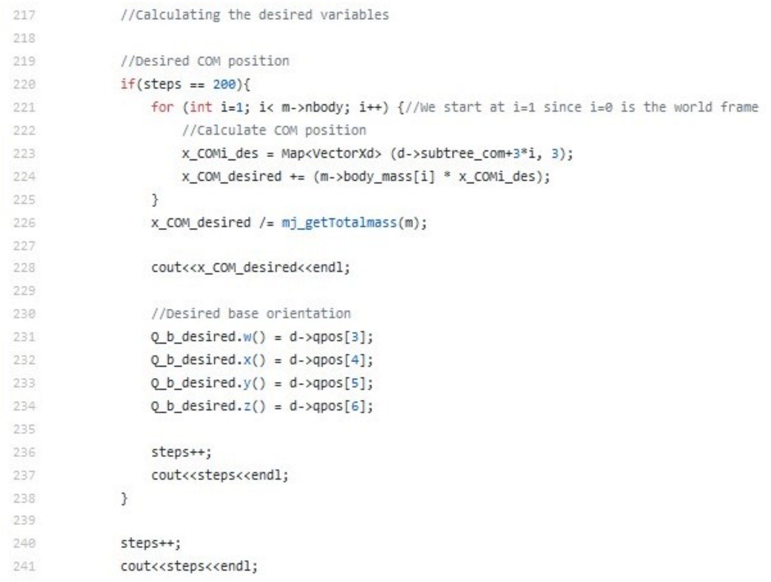
In the easy implementation (see Fig. 12) we then simply compute the Pseudo Inverse of G using a Singular Value Decomposition (SVD) to calculate the contact point wrenches F_C (F_k) according to equation 10.

In a second iteration, we solved for the contact point forces F_C using a quadratic programming framework to solve the Constrained optimization problem as described in section II-B (Eq. 11, 13). The framework called OSQP, which we used, can be found *with this link*. Figure 13 shows our implementation. To be able to use the OSQP framework, we have to convert Eq. 11 to the form of (15) and the linearized constraints (13) to the form of (16)



```
        //3rd step wit tau=J_T*F_k

        MatrixXd J_b_COM = MatrixXd::Zero(3, m->nv); //Base to COM Jacobian in W frame with shape 3x number of degrees of freedom
        MatrixXd J_COM_K = MatrixXd::Zero(number_contact_points*6, m->nv); //Final Com to endeffector Jacobian in W frame with shape 6x number of degrees of freedom
        MatrixXd J_COM_K_T = MatrixXd::Zero(m->nv, number_contact_points*6); //Transpose final Jacobian

        VectorXd tau = VectorXd::Zero(m->nv); //contact wrench

        //Auxiliary variables for final equation 3
        MatrixXd J_COM = MatrixXd::Zero(3, m->nv); //COM Jacobian of all bodies with shape 3x number of degrees of freedom
        MatrixXd J_COMi = MatrixXd::Zero(3, m->nv); //COM Jacobian of body i with shape 3x number of degrees of freedom
        MatrixXd J_b = MatrixXd::Zero(3, m->nv); //Base Jacobian with shape 3x number of degrees of freedom

        MatrixXd J_bk_trans = MatrixXd::Zero(3, m->nv); //Translational part of Com to endeffector k Jacobian with shape 3x number of degrees of freedom
        MatrixXd J_bk_rot = MatrixXd::Zero(3, m->nv); //Rotational part of Com to endeffector k Jacobian with shape 3x number of degrees of freedom
        MatrixXd J_COM_k = MatrixXd::Zero(6, m->nv); //Com to endeffector k Jacobian in W frame with shape 6x number of degrees of freedom
```

Fig. 7: Initialization of the variables for Step 3



```
        //Calculating the desired variables

        //Desired COM position
        if(steps == 200){
            for (int i=1; i< m->nbody; i++) {//We start at i=1 since i=0 is the world frame
                //Calculate COM position
                x_COMi_des = Map<VectorXd> (d->subtree_com+3*i, 3);
                x_COM_desired += (m->body_mass[i] * x_COMi_des);
            }
            x_COM_desired /= mj_getTotalmass(m);

            cout<<x_COM_desired<<endl;

            //Desired base orientation
            Q_b_desired.w() = d->qpos[3];
            Q_b_desired.x() = d->qpos[4];
            Q_b_desired.y() = d->qpos[5];
            Q_b_desired.z() = d->qpos[6];

            steps++;
            cout<<steps<<endl;
        }

        steps++;
        cout<<steps<<endl;
```

Fig. 8: Implementation Step 1: Desired variables

$$\min_x \frac{1}{2}x^T P x + q^T x \tag{15}$$

subject to

$$l \leq Ax \leq u \tag{16}$$

VI. Step 3 Force Mapping

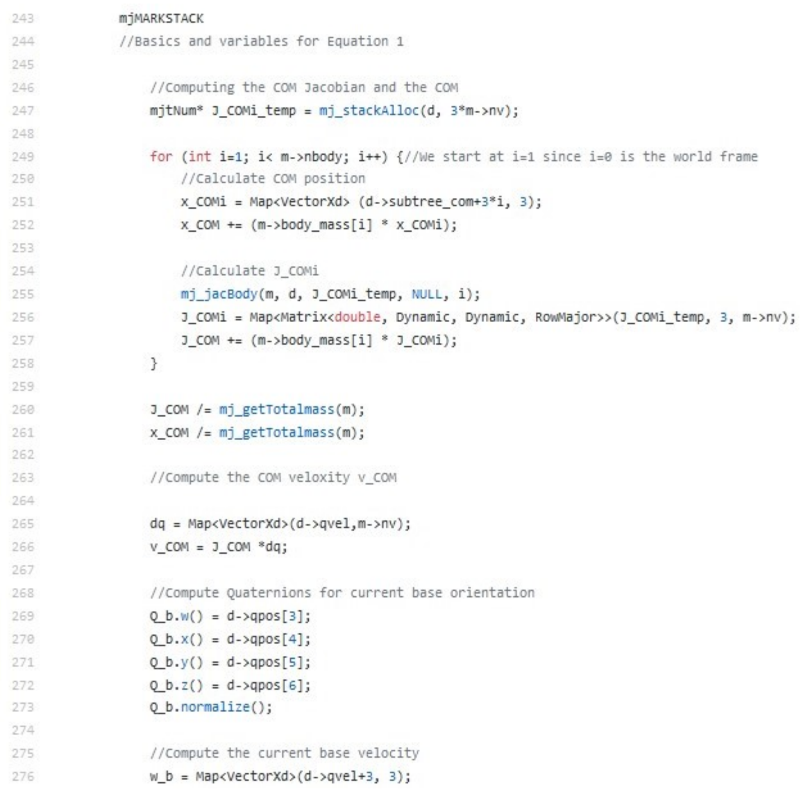
The central part of the 3rd step is the computation of the Jacobian (see Fig. 14). We want to compute the Jacobian J_i from the COM to the frame of the contact points (*J_COM_k*). In order to perform only one matrix multiplication and not a sum as in Eq. 14, we concatenate $J_i \in R^{6\times n}$ to one Jacobian $J \in R^{6m\times n}$ (*F_k*).

In order to get the Jacobian from the COM to the contact point frames, we first compute the Jacobian from the base frame B to the COM (*J_b_COM*) expressed in the world frame W and subtract it then from the translational part of the Jacobian w.r.t the base frame B, which we get as a standard Jacobian from the Mujoco framework (*J_bk_trans*). By performing this subtraction we get the Jacobian w.r.t the COM expressed in the world frame W.

In the last step (see Fig. 15) we then transpose the Jacobian J to finally calculate the control torques τ according to (14), which we then feed into the controller with (*d -> qfrc_applied[i+6]*).

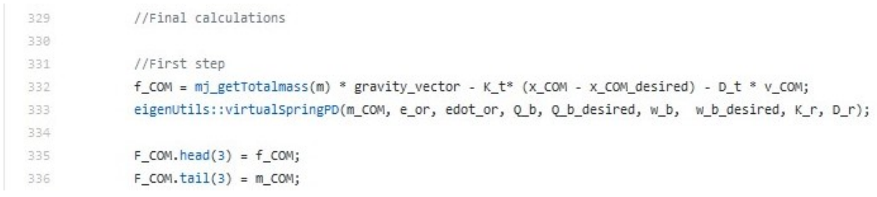
VII. Results and Experiments

As a result from our work, the 12-legged robot Dodo has now the ability to balance against unknown disturbances.



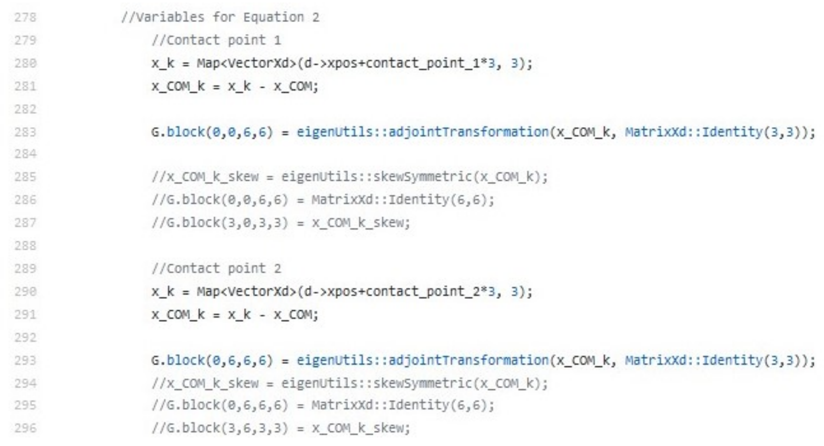
```
mjMARKSTACK
//Basics and variables for Equation 1

    //Computing the COM Jacobian and the COM
    mjtNum* J_COMi_temp = mj_stackAlloc(d, 3*m->nv);

    for (int i=1; i< m->nbody; i++) {//We start at i=1 since i=0 is the world frame
        //Calculate COM position
        x_COMi = Map<VectorXd> (d->subtree_com+3*i, 3);
        x_COM += (m->body_mass[i] * x_COMi);

        //Calculate J_COMi
        mj_jacBody(m, d, J_COMi_temp, NULL, i);
        J_COMi = Map<Matrix<double, Dynamic, Dynamic, RowMajor>>(J_COMi_temp, 3, m->nv);
        J_COM += (m->body_mass[i] * J_COMi);
    }

    J_COM /= mj_getTotalmass(m);
    x_COM /= mj_getTotalmass(m);

    //Compute the COM veloxity v_COM

    dq = Map<VectorXd>(d->qvel,m->nv);
    v_COM = J_COM *dq;

    //Compute Quaternions for current base orientation
    Q_b.w() = d->qpos[3];
    Q_b.x() = d->qpos[4];
    Q_b.y() = d->qpos[5];
    Q_b.z() = d->qpos[6];
    Q_b.normalize();

    //Compute the current base velocity
    w_b = Map<VectorXd>(d->qvel+3, 3);
```

Fig. 9: Implementation Step 1



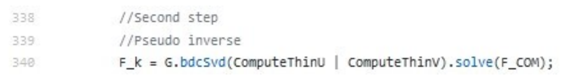
```
//Final calculations

//First step
f_COM = mj_getTotalmass(m) * gravity_vector - K_t* (x_COM - x_COM_desired) - D_t * v_COM;
eigenUtils::virtualSpringPD(m_COM, e_or, edot_or, Q_b, Q_b_desired, w_b,  w_b_desired, K_r, D_r);

F_COM.head(3) = f_COM;
F_COM.tail(3) = m_COM;
```

Fig. 10: Final calculation Step 1



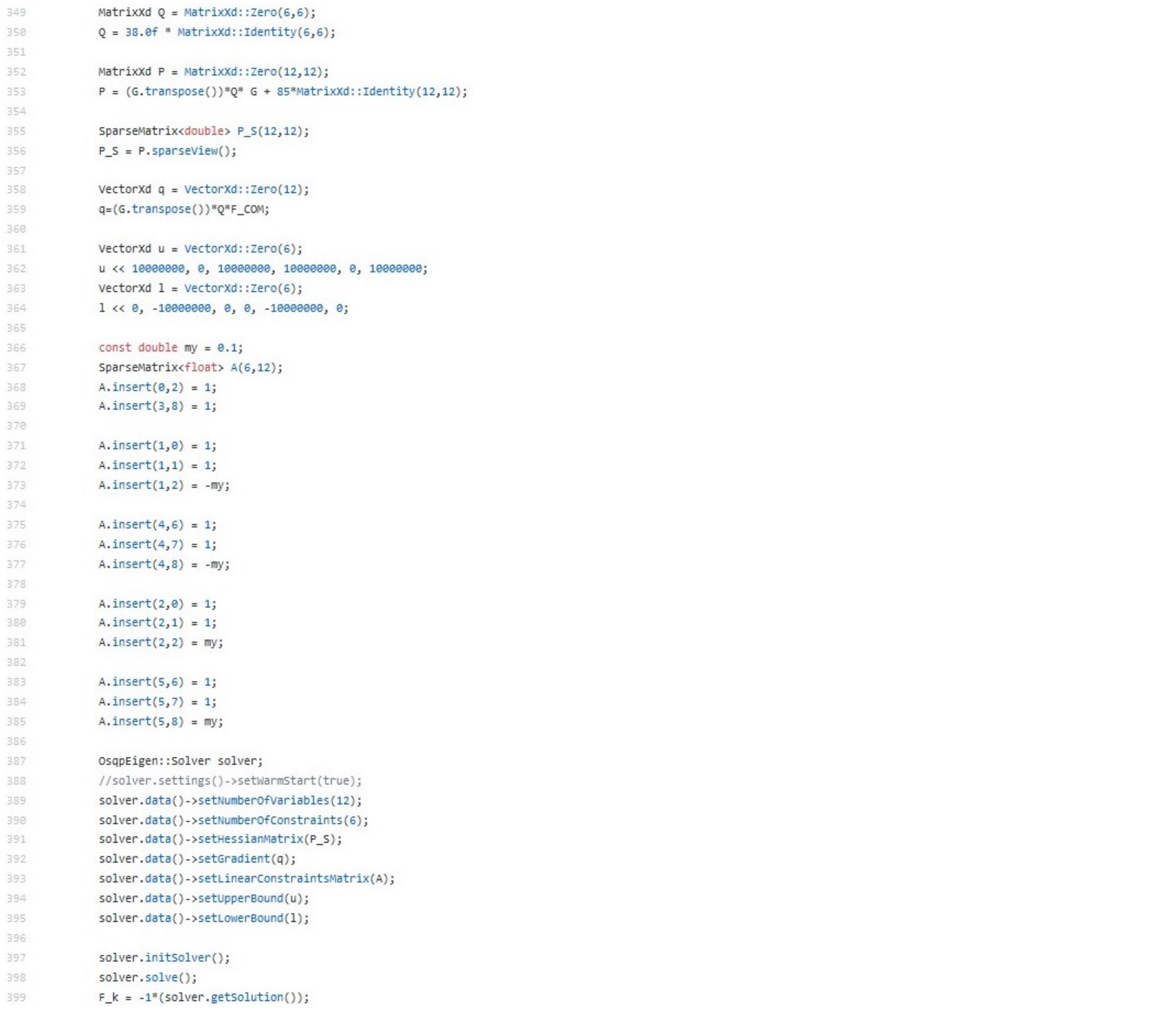
```
//Variables for Equation 2
    //Contact point 1
    x_k = Map<VectorXd>(d->xpos+contact_point_1*3, 3);
    x_COM_k = x_k - x_COM;

    G.block(0,0,6,6) = eigenUtils::adjointTransformation(x_COM_k, MatrixXd::Identity(3,3));

    //x_COM_k_skew = eigenUtils::skewSymmetric(x_COM_k);
    //G.block(0,0,6,6) = MatrixXd::Identity(6,6);
    //G.block(3,0,3,3) = x_COM_k_skew;

    //Contact point 2
    x_k = Map<VectorXd>(d->xpos+contact_point_2*3, 3);
    x_COM_k = x_k - x_COM;

    G.block(0,6,6,6) = eigenUtils::adjointTransformation(x_COM_k, MatrixXd::Identity(3,3));
    //x_COM_k_skew = eigenUtils::skewSymmetric(x_COM_k);
    //G.block(0,6,6,6) = MatrixXd::Identity(6,6);
    //G.block(3,6,3,3) = x_COM_k_skew;
```

Fig. 11: Implementation Step 2



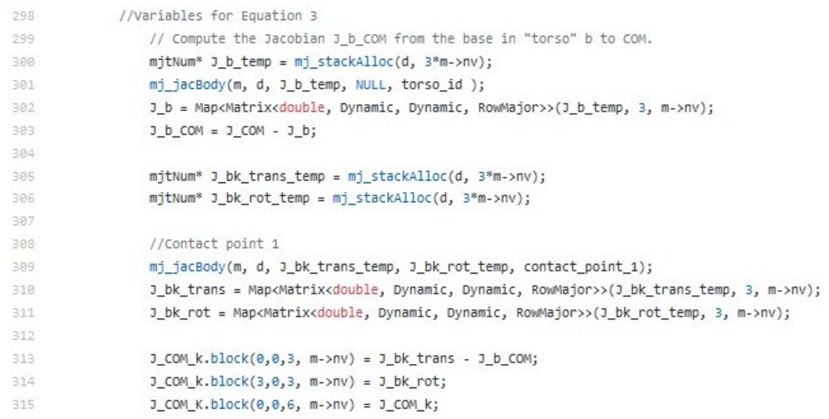
```
//Second step
//Pseudo inverse
F_k = G.bdcSvd(ComputeThinU | ComputeThinV).solve(F_COM);
```

Fig. 12: Final calculation Step 2



```
MatrixXd Q = MatrixXd::Zero(6,6);
Q = 38.0f * MatrixXd::Identity(6,6);

MatrixXd P = MatrixXd::Zero(12,12);
P = (G.transpose())*Q* G + 85*MatrixXd::Identity(12,12);

SparseMatrix<double> P_S(12,12);
P_S = P.sparseView();

VectorXd q = VectorXd::Zero(12);
q=(G.transpose())*Q*F_COM;

VectorXd u = VectorXd::Zero(6);
u << 10000000, 0, 10000000, 10000000, 0, 10000000;
VectorXd l = VectorXd::Zero(6);
l << 0, -10000000, 0, 0, -10000000, 0;

const double my = 0.1;
SparseMatrix<float> A(6,12);
A.insert(0,2) = 1;
A.insert(3,8) = 1;

A.insert(1,0) = 1;
A.insert(1,1) = 1;
A.insert(1,2) = -my;

A.insert(4,6) = 1;
A.insert(4,7) = 1;
A.insert(4,8) = -my;

A.insert(2,0) = 1;
A.insert(2,1) = 1;
A.insert(2,2) = my;

A.insert(5,6) = 1;
A.insert(5,7) = 1;
A.insert(5,8) = my;

OsqpEigen::Solver solver;
//solver.settings()->setWarmStart(true);
solver.data()->setNumberOfVariables(12);
solver.data()->setNumberOfConstraints(6);
solver.data()->setHessianMatrix(P_S);
solver.data()->setGradient(q);
solver.data()->setLinearConstraintsMatrix(A);
solver.data()->setUpperBound(u);
solver.data()->setLowerBound(l);

solver.initSolver();
solver.solve();
F_k = -1*(solver.getSolution());
```

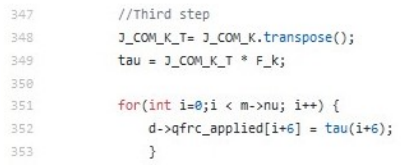
Fig. 13: Implementation Optimization



```
//Variables for Equation 3
    // Compute the Jacobian J_b_COM from the base in "torso" b to COM.
    mjtNum* J_b_temp = mj_stackAlloc(d, 3*m->nv);
    mj_jacBody(m, d, J_b_temp, NULL, torso_id );
    J_b = Map<Matrix<double, Dynamic, Dynamic, RowMajor>>(J_b_temp, 3, m->nv);
    J_b_COM = J_COM - J_b;

    mjtNum* J_bk_trans_temp = mj_stackAlloc(d, 3*m->nv);
    mjtNum* J_bk_rot_temp = mj_stackAlloc(d, 3*m->nv);

    //Contact point 1
    mj_jacBody(m, d, J_bk_trans_temp, J_bk_rot_temp, contact_point_1);
    J_bk_trans = Map<Matrix<double, Dynamic, Dynamic, RowMajor>>(J_bk_trans_temp, 3, m->nv);
    J_bk_rot = Map<Matrix<double, Dynamic, Dynamic, RowMajor>>(J_bk_rot_temp, 3, m->nv);

    J_COM_k.block(0,0,3, m->nv) = J_bk_trans - J_b_COM;
    J_COM_k.block(3,0,3, m->nv) = J_bk_rot;
    J_COM_K.block(0,0,6, m->nv) = J_COM_k;
```

Fig. 14: Implementation Step 3

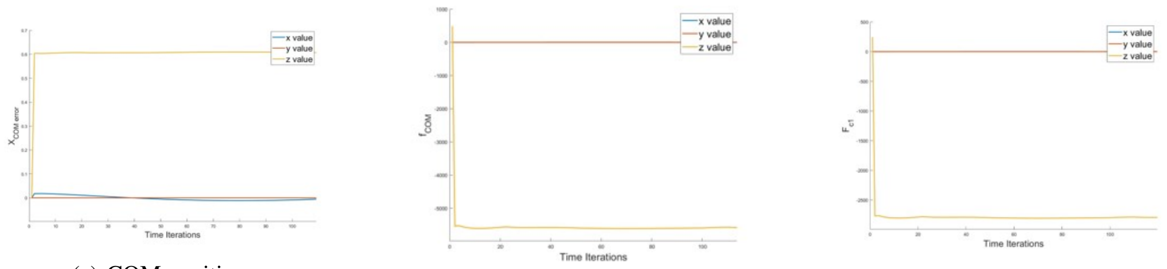
Solving for the contact forces F_C without the optimization but just using the Pseudo Inverse of G, the robot after some time ends up in a stable but suboptimal pose, with X-shaped legs (see Fig. 17a).



```
//Third step
J_COM_K_T= J_COM_K.transpose();
tau = J_COM_K_T * F_k;

for(int i=0;i < m->nu; i++) {
    d->qfrc_applied[i+6] = tau(i+6);
    }
```

Fig. 15: Final calculation Step 3

In order to check the internal system state for the pure case of balancing of the x-shaped Dodo, we have visualized the COM position error (see Fig. 16a), the COM force components (see Fig. 16b) and the contact force components (see Fig. 16c). We see, that we only have non-zero force components in z-direction, which counteract the influence of gravity.



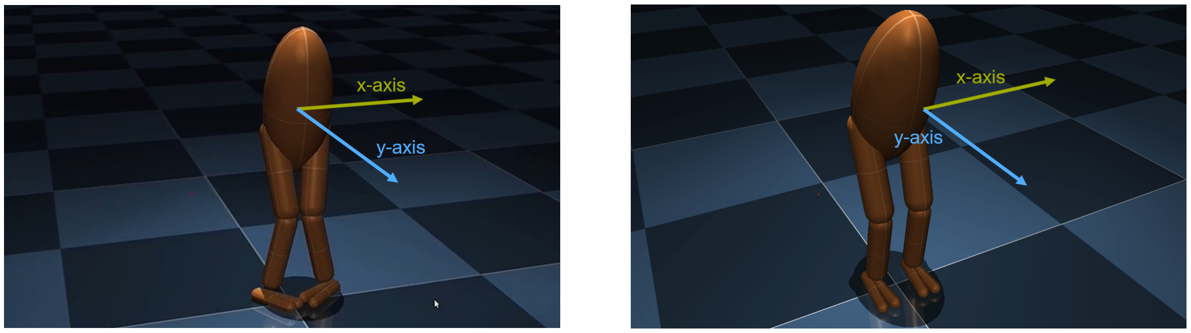
(a) COM position error

(b) COM force components

(c) Contact force components

Fig. 16: The internal system state of the x-shaped Dodo for case of pure balancing without external disturbances

We have tested the stability of the x-shaped robot against forces in x- and y- direction and a torque around the x-axis (see Fig. 17a). As a result, the robot can withstand a force of 51 N in x-direction, a force of 193 N in y direction and a torque of 36Nm around the x-axis.



(a) Dodo with a x-shaped posture.

(b) Straight-legged Dodo with constrained optimization.

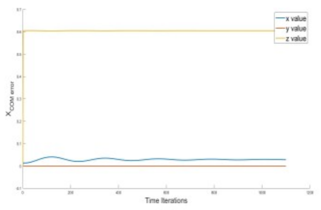
Fig. 17: X-shaped and straight-legged Dodo. The x- and y- axis indicate the axis, where we applied disturbance forces and a torque around the x-axis for testing the stability of Dodo.

Using the optimization approach and tuning the parameters for Q and α the robot results in a straight-legged pose 17b. But this doesn't imply a more stable pose. The pose gets more straight but rather unstable with a higher α, as we minimize the L2-norm of the contact forces in (11). Therefore choosing this parameter we have to trade-off between stability against external disturbances and a straight and light pose with lower torques we have to apply. After choosing $\alpha = 85$ and $Q = 38.0$ (see Fig. 13), we have first tested again the internal state of the straight-legged robot for the pure case of balancing, with the COM position error (see Fig. 18a), the COM force components (see Fig. 18b) and the contact force components (see Fig. 18c). For the contact force components we see clearly, that the z-component is significantly smaller (-1300N) than without the constrained optimization (-2600N).

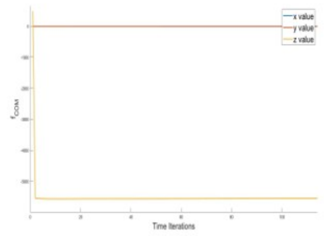
Then the straight-legged robot with the optimization approach was tested against external disturbances. It can withstand a force of 27N in x-direction, a force of 215N in y-direction and a torque of 9Nm around the x-axis.

VIII. Conclusion

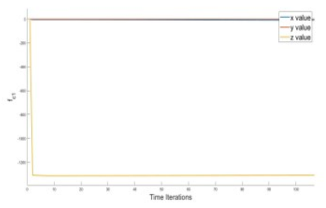
Finally one can say, that our torque based control strategy for biped balancing results in an efficient balancing of the Dodo. The approach we followed allows to recover position and posture, without having to compute inverse dynamics or kinematics or having to measure contact forces. We validated our implementation via simulation.



(a) COM position error



(b) COM force components



(c) Contact force components

Fig. 18: The internal system state of the straight-legged Dodo with constrained optimization for case of pure balancing without external disturbances

An extension for future work would be to apply the biped robot in situations with more contacts (here only 2), or situations, where the support phases vary (single or double support) or to even realize a walking strategy of the robot.

Bibliography

[1] S. Setiawan, J. Yamaguchi, S. Hyon, and A. Takanishi, "Physical interaction between human and a bipedal humanoid robot - realization of human-follow walking," English, *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 1, pp. 361–367, Jan. 1999, Proceedings of the 1999 IEEE International Conference on Robotics and Automation, ICRA99 ; Conference date: 10-05-1999 Through 15-05-1999, ISSN: 1050-4729.

[2] S.-H. Lee and A. Goswami, "Ground reaction force control at each foot: A momentum-based humanoid balance controller for non-level and non-stationary ground," Nov. 2010, pp. 3157–3162. DOI: 10.1109/IROS.2010.5650416.

[3] C. Ott, M. A. Roa, and G. Hirzinger, "Posture and balance control for biped robots based on contact force optimization," in *2011 11th IEEE-RAS International Conference on Humanoid Robots*, IEEE, 2011, pp. 26–33.

[4] B. Henze, M. A. Roa, and C. Ott, "Passivity-based whole-body balancing for torque-controlled humanoid robots in multi-contact scenarios," *The International Journal of Robotics Research*, vol. 35, no. 12, pp. 1522–1543, 2016.

[5] C. Ott and S.-H. Hyon, "Torque-based balancing," in *Humanoid Robotics: A Reference*, A. Goswami and P. Vadakkepat, Eds. Dordrecht: Springer Netherlands, 2019, pp. 1361–1386, ISBN: 978-94-007-6046-2. DOI: 10.1007/978-94-007-6046-2_39. [Online]. Available: https://doi.org/10.1007/978-94-007-6046-2_39.

[6] J. Pratt and B. Krupp, "Design of a bipedal walking robot," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 6962, Apr. 2008. DOI: 10.1117/12.777973.

[7] A. Albu-Schäffer, C. Ott, and G. Hirzinger, "A unified passivity-based control framework for position, torque and impedance control of flexible joint robots," *The international journal of robotics research*, vol. 26, no. 1, pp. 23–39, 2007.