Task-Space Inverse Dynamics: Implementation

Quadratic-Programming based Control for Legged Robots

Andrea Del Prete

University of Trento

Table of contents

Introduction

Introduction

This document explains the implementation of the control framework Task-Space Inverse Dynamics (TSID).

[1] https://github.com/stack-of-tasks/tsid

Introduction

This document explains the implementation of the control framework Task-Space Inverse Dynamics (TSID).

To simplify the job we rely on the open-source C++ library TSID[1].

[1] https://github.com/stack-of-tasks/tsid

Introduction

This document explains the implementation of the control framework Task-Space Inverse Dynamics (TSID).

To simplify the job we rely on the open-source C++ library TSID[1].

TSID (currently) relies on:

- Eigen for linear algebra
- Pinocchio for multi-body dynamics computations
- Eiquadprog for solving Quadratic Programs

[1] https://github.com/stack-of-tasks/tsid

Main features: Pros & Cons

CONS

- Not mature (Feb 2017)
- Many missing features you may need for your application
  - Hierarchy
  - Fixed-base robots
  - Joint pos-vel limits
  - Actuation limits
  - Bilateral contacts
  - Line contacts
  - ...

Main features: Pros & Cons

CONS

- Not mature (Feb 2017)
- Many missing features you may need for your application
  - Hierarchy
  - Fixed-base robots
  - Joint pos-vel limits
  - Actuation limits
  - Bilateral contacts
  - Line contacts
  - ...

PROS

- Efficient ($<$0.6 ms for humanoid)
- Tested in simulation & on HRP-2
- Open source
- Modular design
  - $\rightarrow$ easy to extend
- Python bindings
- No alternative (AFAIK)

Key Concepts

Task

- Motion
- Force
- Actuation

Key Concepts

Task

- Motion
- Force
- Actuation

Rigid Contact

- similar to Task, but
- associated to reaction forces

Key Concepts

Task

- Motion
- Force
- Actuation

Rigid Contact

- similar to Task, but
- associated to reaction forces

Inverse Dynamics Formulation

- collects `Tasks` and `RigidContacts`
- translates them into HQP

Key Concepts

Task

- Motion
- Force
- Actuation

Rigid Contact

- similar to Task, but
- associated to reaction forces

Inverse Dynamics Formulation

- collects `Tasks` and `RigidContacts`
- translates them into HQP

HQP Solver

- solves a HQP

Other Concepts

Constraint

- affine function
- purely mathematical
- used to represent HQP

Other Concepts

Constraint

- affine function
- purely mathematical
- used to represent HQP

Robot Wrapper

- contains robot model
- provides utility functions to compute robot quantities
- e.g., mass matrix, Jacobians

Other Concepts

Constraint

- affine function
- purely mathematical
- used to represent HQP

Robot Wrapper

- contains robot model
- provides utility functions to compute robot quantities
- e.g., mass matrix, Jacobians

Trajectory

- maps time to vector values
- pos, vel, acc
- position and velocity can have different sizes (Lie groups)

Details

Robot Wrapper 1/2

Interface for computing robot-related quantities:

```
RobotWrapper(string filename, vector<string> package_dirs,
             JointModelVariant rootJoint);

int nq(); // size of configuration vector q
int nv(); // size of velocity vector v

Model & model(); // reference to robot model (Pinocchio)

// Compute all quantities and store them into data
void computeAllTerms(Data &data, Vector q, Vector v);
```

Robot Wrapper 2/2

```
Vector rotor_inertias();
Vector gear_ratios();

Vector3 com(Data data);
Vector3 com_vel(Data data);
Vector3 com_acc(Data data);
Matrix3x Jcom(Data data);

Matrix mass(Data data);
Vector nonLinearEffects(Data data);

SE3 position(Data data, JointIndex index);
Motion velocity(Data data, JointIndex index);
Motion acceleration(Data data, JointIndex index);
Matrix6x jacobianWorld(Data data, JointIndex index);
Matrix6x jacobianLocal(Data data, JointIndex index);
```

ConstraintBase

- A linear (affine) function

ConstraintBase

- A linear (affine) function
- Purely mathematical object

ConstraintBase

- A linear (affine) function
- Purely mathematical object
- “Unaware” of what the function represents

Three kinds of constraints:

ConstraintBase

- A linear (affine) function
- Purely mathematical object
- “Unaware” of what the function represents

Three kinds of constraints:

- Equalities, represented by matrix A and vector a:

$$Ax = a$$

ConstraintBase

- A linear (affine) function
- Purely mathematical object
- “Unaware” of what the function represents

Three kinds of constraints:

- Equalities, represented by matrix A and vector a:

$$Ax = a$$

- Inequalities, represented by matrix A and vectors lb and ub:

$$lb \leq Ax \leq ub$$

ConstraintBase

- A linear (affine) function
- Purely mathematical object
- "Unaware" of what the function represents

Three kinds of constraints:

- Equalities, represented by matrix A and vector a:

$$Ax = a$$

- Inequalities, represented by matrix A and vectors lb and ub:

$$lb \leq Ax \leq ub$$

- Bounds, represented by vectors lb and ub:

$$lb \leq x \leq ub$$

ConstraintBase

```
ConstraintBase(string name, int rows, int cols);

bool isEquality();
bool isInequality();
bool isBound();

Matrix matrix();
Vector vector();
Vector lowerBound();
Vector upperBound();

bool setMatrix(Matrix A);
bool setVector(Vector b);
bool setLowerBound(Vector lb);
bool setUpperBound(Vector ub);

bool checkConstraint(Vector x);
```

TaskBase

Interface of `TaskBase`:

```
TaskBase(string name, Model model);
Constraint compute(double t, Vector q, Vector v, Data data);
```

TaskBase

Interface of `TaskBase`:

```
TaskBase(string name, Model model);
Constraint compute(double t, Vector q, Vector v, Data data);
```

Three kinds of task:

- `TaskMotion`: linear function of robot accelerations
- `TaskContactForce`: linear function of contact forces
- `TaskActuation`: linear function of joint torques

TaskBase

Interface of `TaskBase`:

```
TaskBase(string name, Model model);
Constraint compute(double t, Vector q, Vector v, Data data);
```

Three kinds of task:

- `TaskMotion`: linear function of robot accelerations
- `TaskContactForce`: linear function of contact forces
- `TaskActuation`: linear function of joint torques

Tasks can compute either:

- equality constraints, e.g., `TaskComEquality`, `TaskJointPosture`, `TaskSE3Equality`

TaskBase

Interface of `TaskBase`:

```
TaskBase(string name, Model model);
Constraint compute(double t, Vector q, Vector v, Data data);
```

Three kinds of task:

- `TaskMotion`: linear function of robot accelerations
- `TaskContactForce`: linear function of contact forces
- `TaskActuation`: linear function of joint torques

Tasks can compute either:

- equality constraints, e.g., `TaskComEquality`, `TaskJointPosture`, `TaskSE3Equality`
- bounds, e.g., `TaskJointBounds` (not implemented yet)

TaskBase

Interface of `TaskBase`:

```
TaskBase(string name, Model model);
Constraint compute(double t, Vector q, Vector v, Data data);
```

Three kinds of task:

- `TaskMotion`: linear function of robot accelerations
- `TaskContactForce`: linear function of contact forces
- `TaskActuation`: linear function of joint torques

Tasks can compute either:

- equality constraints, e.g., `TaskComEquality`, `TaskJointPosture`, `TaskSE3Equality`
- bounds, e.g., `TaskJointBounds` (not implemented yet)
- inequality constraints, e.g., friction cones

ContactBase

Interface of `ContactBase`:

```
ContactBase(name, Kp, Kd, bodyName, regWeight);
ConstraintBase computeMotionTask(t, q, v, data);
InequalityConstraint computeForceTask(t, q, v, data);
ConstraintBase computeForceRegularizationTask(t, q, v, data);
Matrix computeForceGeneratorMatrix();
```

ContactBase

Interface of `ContactBase`:

```
ContactBase(name, Kp, Kd, bodyName, regWeight);
ConstraintBase computeMotionTask(t, q, v, data);
InequalityConstraint computeForceTask(t, q, v, data);
ConstraintBase computeForceRegularizationTask(t, q, v, data);
Matrix computeForceGeneratorMatrix();
```

Motion task:

- represents motion constraint caused by rigid contact
- $J\dot{v}_q = -\dot{J}v_q$

ContactBase

Interface of `ContactBase`:

```
ContactBase(name, Kp, Kd, bodyName, regWeight);
ConstraintBase computeMotionTask(t, q, v, data);
InequalityConstraint computeForceTask(t, q, v, data);
ConstraintBase computeForceRegularizationTask(t, q, v, data);
Matrix computeForceGeneratorMatrix();
```

Motion task:

- represents motion constraint caused by rigid contact
- $J\dot{v}_q = -\dot{J}v_q - K_p e - K_d \dot{e}$

ContactBase

Interface of `ContactBase`:

```
ContactBase(name, Kp, Kd, bodyName, regWeight);
ConstraintBase computeMotionTask(t, q, v, data);
InequalityConstraint computeForceTask(t, q, v, data);
ConstraintBase computeForceRegularizationTask(t, q, v, data);
Matrix computeForceGeneratorMatrix();
```

Motion task:

- represents motion constraint caused by rigid contact
- $J\dot{v}_q = -\dot{J}v_q - K_p e - K_d \dot{e}$

Force task:

- represents inequality constraints acting on contact forces
- e.g., friction cone constraints
- $Af \leq a$

ContactBase

Interface of `ContactBase`:

```
ContactBase(name, Kp, Kd, bodyName, regWeight);
ConstraintBase computeMotionTask(t, q, v, data);
InequalityConstraint computeForceTask(t, q, v, data);
ConstraintBase computeForceRegularizationTask(t, q, v, data);
Matrix computeForceGeneratorMatrix();
```

Force Regularization task:

- regularizes contact forces
- e.g., keep them close to friction cone center

ContactBase

Interface of `ContactBase`:

```
ContactBase(name, Kp, Kd, bodyName, regWeight);
ConstraintBase computeMotionTask(t, q, v, data);
InequalityConstraint computeForceTask(t, q, v, data);
ConstraintBase computeForceRegularizationTask(t, q, v, data);
Matrix computeForceGeneratorMatrix();
```

Force Regularization task:

- regularizes contact forces
- e.g., keep them close to friction cone center

Force-Generator matrix T:

- maps force variables to motion constraint representation
- Dynamic: $M\dot{v}_q + h = S^\top \tau + J^\top T f$
- Motion constraint: $J\dot{v}_q = -\dot{J} v_q$
- Friction cones: $Af \leq a$

Contact6d

- unilateral plane contact (polygonal shape)

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

PROBLEM

- minimal force representation would be 6d (3d force + 3d moment)

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

PROBLEM

- minimal force representation would be 6d (3d force + 3d moment)
- hard to write friction constraints with 6d force representation (especially for non-rectangular contact shapes)

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

PROBLEM

- minimal force representation would be 6d (3d force + 3d moment)
- hard to write friction constraints with 6d force representation (especially for non-rectangular contact shapes)
- easy to write friction constraints if reaction force represented as collection of 3d forces applied at vertices of contact surface

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

PROBLEM

- minimal force representation would be 6d (3d force + 3d moment)
- hard to write friction constraints with 6d force representation (especially for non-rectangular contact shapes)
- easy to write friction constraints if reaction force represented as collection of 3d forces applied at vertices of contact surface
  - redundant representation, e.g., 4-vertex surface $\rightarrow$ 12 variables

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

PROBLEM

- minimal force representation would be 6d (3d force + 3d moment)
- hard to write friction constraints with 6d force representation (especially for non-rectangular contact shapes)
- easy to write friction constraints if reaction force represented as collection of 3d forces applied at vertices of contact surface
  - redundant representation, e.g., 4-vertex surface $\rightarrow$ 12 variables
- redundancy is an issue for motion constraint if HQP solver does not handle redundant constraints (as `eiQuadProg`).

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

PROBLEM

- minimal force representation would be 6d (3d force + 3d moment)
- hard to write friction constraints with 6d force representation (especially for non-rectangular contact shapes)
- easy to write friction constraints if reaction force represented as collection of 3d forces applied at vertices of contact surface
  - redundant representation, e.g., 4-vertex surface $\rightarrow$ 12 variables
- redundancy is an issue for motion constraint if HQP solver does not handle redundant constraints (as `eiQuadProg`).

SOLUTION

- use 6d representation for motion constraint $J\dot{v}_q = -\dot{J}v_q \in \mathbb{R}^6$

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

PROBLEM

- minimal force representation would be 6d (3d force + 3d moment)
- hard to write friction constraints with 6d force representation (especially for non-rectangular contact shapes)
- easy to write friction constraints if reaction force represented as collection of 3d forces applied at vertices of contact surface
  - redundant representation, e.g., 4-vertex surface $\rightarrow$ 12 variables
- redundancy is an issue for motion constraint if HQP solver does not handle redundant constraints (as `eiQuadProg`).

SOLUTION

- use 6d representation for motion constraint $J\dot{v}_q = -\dot{J}v_q \in \mathbb{R}^6$
- but 12d representation for force variable $f \in \mathbb{R}^{12}$

Contact6d

- unilateral plane contact (polygonal shape)
- 6d motion constraint $\rightarrow$ no motion allowed in any direction

PROBLEM

- minimal force representation would be 6d (3d force + 3d moment)
- hard to write friction constraints with 6d force representation (especially for non-rectangular contact shapes)
- easy to write friction constraints if reaction force represented as collection of 3d forces applied at vertices of contact surface
    - redundant representation, e.g., 4-vertex surface $\rightarrow$ 12 variables
- redundancy is an issue for motion constraint if HQP solver does not handle redundant constraints (as `eiQuadProg`).

SOLUTION

- use 6d representation for motion constraint $J\dot{v}_q = -\dot{J}v_q \in \mathbb{R}^6$
- but 12d representation for force variable $f \in \mathbb{R}^{12}$
- force-generator matrix $T \in \mathbb{R}^{6\times 12}$ defines mapping between two representations: $\tau_{contact} = J^\top Tf$

InverseDynamicsFormulationBase

Central class of the whole library

InverseDynamicsFormulationBase

Central class of the whole library

Methods to add tasks:

```
addMotionTask(MotionTask task, double weight, int priority);
addForceTask(ForceTask task, double weight, int priority);
addTorqueTask(TorqueTask task, double weight, int priority);
```

InverseDynamicsFormulationBase

Central class of the whole library

Methods to add tasks:

```
addMotionTask(MotionTask task, double weight, int priority);
addForceTask(ForceTask task, double weight, int priority);
addTorqueTask(TorqueTask task, double weight, int priority);
```

Method to add rigid contacts:

```
addRigidContact(RigidContact contact);
```

InverseDynamicsFormulationBase

Central class of the whole library

Methods to add tasks:

```
addMotionTask(MotionTask task, double weight, int priority);
addForceTask(ForceTask task, double weight, int priority);
addTorqueTask(TorqueTask task, double weight, int priority);
```

Method to add rigid contacts:

```
addRigidContact(RigidContact contact);
```

Methods to convert TSID problem into (Hierarchical) QP:

```
HqpData computeProblemData(double time, Vector q, Vector v);
```

InverseDynamicsFormulationBase

Central class of the whole library

Methods to add tasks:

```
addMotionTask(MotionTask task, double weight, int priority);
addForceTask(ForceTask task, double weight, int priority);
addTorqueTask(TorqueTask task, double weight, int priority);
```

Method to add rigid contacts:

```
addRigidContact(RigidContact contact);
```

Methods to convert TSID problem into (Hierarchical) QP:

```
HqpData computeProblemData(double time, Vector q, Vector v);
```

HqpData defined as:

```
#typedef vector<pair<double, ConstraintBase>> ConstraintLevel
#typedef vector<ConstraintLevel> HqpData
```

HQP Solvers

Using `InverseDynamicsFormulationBase` you get an `HqpData` object.

HQP Solvers

Using `InverseDynamicsFormulationBase` you get an `HqpData` object.

Then you need to solve this HQP.

HQP Solvers

Using `InverseDynamicsFormulationBase` you get an `HqpData` object.

Then you need to solve this HQP.

All HQP solvers implement this interface (`SolverHQPBase`):

```
void resize(int nVar, int nEq, int nIn);
HqpOutput solve(HqpData data);
```

HQP Solvers

Using `InverseDynamicsFormulationBase` you get an `HqpData` object.

Then you need to solve this HQP.

All HQP solvers implement this interface (`SolverHQPBase`):

```
void resize(int nVar, int nEq, int nIn);
HqpOutput solve(HqpData data);
```

`HqpOutput` is defined as:

```
class HqpOutput
{
     QpStatusFlag flag;
     Vector x, lambda;
}
```

Available HQP Solvers

- Several solvers currently implemented

Available HQP Solvers

- Several solvers currently implemented
- None of them supports hierarchy

Available HQP Solvers

- Several solvers currently implemented
- None of them supports hierarchy
- $\rightarrow$ HQP problems can only have two hierarchy levels.

Available HQP Solvers

- Several solvers currently implemented
- None of them supports hierarchy
- $\rightarrow$ HQP problems can only have two hierarchy levels.
- All solvers based on `EiQuadProg`: a modified version of `uQuadProg++` working with `Eigen`

Available HQP Solvers

- Several solvers currently implemented
- None of them supports hierarchy
- $\rightarrow$ HQP problems can only have two hierarchy levels.
- All solvers based on `EiQuadProg`: a modified version of `uQuadProg++` working with `Eigen`
- To improve efficiency, two optimized versions have been developed:

Available HQP Solvers

- Several solvers currently implemented
- None of them supports hierarchy
- $\rightarrow$ HQP problems can only have two hierarchy levels.
- All solvers based on `EiQuadProg`: a modified version of `uQuadProg++` working with `Eigen`
- To improve efficiency, two optimized versions have been developed:
  - `EiquadprogRealTime`: the fastest, but matrix sizes known at compile time

Available HQP Solvers

- Several solvers currently implemented
- None of them supports hierarchy
- $\rightarrow$ HQP problems can only have two hierarchy levels.
- All solvers based on `EiQuadProg`: a modified version of `uQuadProg++` working with `Eigen`
- To improve efficiency, two optimized versions have been developed:
  - `EiquadprogRealTime`: the fastest, but matrix sizes known at compile time
  - `EiquadprogFast`: dynamic matrix sizes (memory allocation performed only when resizing)

Available HQP Solvers

- Several solvers currently implemented
- None of them supports hierarchy
- $\rightarrow$ HQP problems can only have two hierarchy levels.
- All solvers based on `EiQuadProg`: a modified version of `uQuadProg++` working with `Eigen`
- To improve efficiency, two optimized versions have been developed:
  - `EiquadprogRealTime`: the fastest, but matrix sizes known at compile time
  - `EiquadprogFast`: dynamic matrix sizes (memory allocation performed only when resizing)

Results on HRP-2's computer (very old):

```
60 variables, 18 equalities, 40 inequalities
*** PROFILING RESULTS [ms]     (min - avg - max ) ***
Eiquadprog .................. 0.651 0.704 0.870
Eiquadprog Fast ............. 0.563 0.605 0.810
Eiquadprog Real Time ........ 0.543 0.592 0.712

active inequalities .... 16.0 19.8 26.0
```

Python Example

Description

- Code snippets

Description

- Code snippets
- Biped robot with both feet on the ground (double support)

Description

- Code snippets
- Biped robot with both feet on the ground (double support)
- Control center of mass (CoM) for balance

Description

- Code snippets
- Biped robot with both feet on the ground (double support)
- Control center of mass (CoM) for balance
- Control joint angles (posture) for whole-body stability

Description

- Code snippets
- Biped robot with both feet on the ground (double support)
- Control center of mass (CoM) for balance
- Control joint angles (posture) for whole-body stability
- Good starting point before moving to more complex scenarios

Create Robot Wrapper

```
import pinocchio as se3
from tsid import RobotWrapper, ...

path = '/../models/romeo'
urdf = path + '/urdf/romeo.urdf'
vec = se3.StdVec_StdString()
vec.extend(item for item in path)
robot = RobotWrapper(urdf, vec, se3.JointModelFreeFlyer(),
                     False)
```

Create Inverse Dynamics Formulation

```
invdyn = InverseDynamicsFormulationAccForce("tsid", robot,
                                             False)

q = getNeutralConfigurationFromSrdf(robot.model(), srdf, False)
v = matlib.zeros(robot.nv).T
invdyn.computeProblemData(t, q, v)
data = invdyn.data()
```

Create Contact

```
contactRF = Contact6d("contact_rfoot", robot, rf_frame_name,
                        contact_points, contact_normal, mu,
                        fMin, fMax, w_forceReg)

contactRF.setKp(kp_contact * matlib.ones(6).T)
contactRF.setKd(2*sqrt(kp_contact) * matlib.ones(6).T)

rf_joint_id = robot.model().getJointId(rf_frame_name)
H_rf_ref = robot.position(data, rf_joint_id)
contactRF.setReference(H_rf_ref)

invdyn.addRigidContact(contactRF)

# repeat for other contact(s)
```

Create Center-of-Mass Task

```
comTask = TaskComEquality("task-com", robot)

comTask.setKp(kp_com * matlib.ones(3).T)
comTask.setKd(2*sqrt(kp_com) * matlib.ones(3).T)

invdyn.addMotionTask(comTask, w_com, 1, 0.0)
```

Create Posture Task

```
postureTask = TaskJointPosture("task-posture", robot)

postureTask.setKp(kp_posture*matlib.ones(robot.nv-6).T)
postureTask.setKd(2*sqrt(kp_posture)*matlib.ones(robot.nv-6).T)

invdyn.addMotionTask(postureTask, w_posture, 1, 0.0)
```

Create Reference Task Trajectories

```
com_ref = robot.com(data)
trajCom = TrajectoryEuclidianConstant("traj_com", com_ref)

q_ref = q[7:]
trajPosture = TrajectoryEuclidianConstant("traj_joint", q_ref)
```

Create HQP Solver

```
solver = SolverHQuadProg("qp solver")
solver.resize(invdyn.nVar, invdyn.nEq, invdyn.nIn)
```

Control Loop

```
for i in range(0, N_SIMULATION_STEPS):
    comTask.setReference(trajCom.computeNext())
    postureTask.setReference(trajPosture.computeNext())

    # get current state estimation
    (q, v) = ...

    HQPData = invdyn.computeProblemData(t, q, v)

    sol = solver.solve(HQPData)
    tau = invdyn.getActuatorForces(sol)

    # send desired joint torques (tau) to actuators
    ...
```

Simulation Loop

```
for i in range(0, N_SIMULATION_STEPS):
    ...

    # assuming perfect torque-acceleration tracking...
    dv = invdyn.getAccelerations(sol)

    # integrate desired accelerations
    q = se3.integrate(robot.model(), q, dt*v)
    v += dt*dv

    # increase time
    t += dt
```

Exercises

Exercise 1: CoM Set-Point Regulation

- Run provided example (`tsid/exercizes/ex_1.py`) and check that the robot does not move

Exercise 1: CoM Set-Point Regulation

- Run provided example (`tsid/exercizes/ex_1.py`) and check that the robot does not move
- Change references of CoM/posture and look what happens

Exercise 1: CoM Set-Point Regulation

- Run provided example (`tsid/exercizes/ex_1.py`) and check that the robot does not move
- Change references of CoM/posture and look what happens
- Change CoM/posture gains and see effect

Exercise 1: CoM Set-Point Regulation

- Run provided example (`tsid/exercizes/ex_1.py`) and check that the robot does not move
- Change references of CoM/posture and look what happens
- Change CoM/posture gains and see effect
- Change CoM/posture weights and see effect

Exercise 1: CoM Set-Point Regulation

- Run provided example (`tsid/exercizes/ex_1.py`) and check that the robot does not move
- Change references of CoM/posture and look what happens
- Change CoM/posture gains and see effect
- Change CoM/posture weights and see effect
- Set reference CoM outside support polygon (e.g., 20 cm to the side), what happens? Why?

Exercise 2: CoM Sinusoidal Tracking

- Run provided code (`tsid/exercizes/ex_2.py`) and check the sinusoidal reference CoM tracking

Exercise 2: CoM Sinusoidal Tracking

- Run provided code (`tsid/exercizes/ex_2.py`) and check the sinusoidal reference CoM tracking
- Increase CoM frequency until tracking gets bad. Why does that happen?

Exercise 2: CoM Sinusoidal Tracking

- Run provided code (`tsid/exercizes/ex_2.py`) and check the sinusoidal reference CoM tracking
- Increase CoM frequency until tracking gets bad. Why does that happen?
- Set contact feedback gains to zero, what happens? Why?

Exercise 2: CoM Sinusoidal Tracking

- Run provided code (`tsid/exercizes/ex_2.py`) and check the sinusoidal reference CoM tracking
- Increase CoM frequency until tracking gets bad. Why does that happen?
- Set contact feedback gains to zero, what happens? Why?
- Add contact on hand

Exercise 3: Taking a Step

- Extend code to make robot take a step (solution in `tsid/demo/demo_romeo.py`)