Stability polygon Documentation

Release 1.0.0

Hervé Audren

Dec 11, 2017

Contents

1 Recursive Projection **3**
- 1.1 Description of the algorithm . . . 3
- 1.2 How to use this class . . . 4
- 1.3 Class API . . . 4

2 Stability Polygon and contacts **7**

3 Linear Projection **9**
- 3.1 Principle . . . 9
- 3.2 Example of usage . . . 9
- 3.3 Class API . . . 10

4 Backends **11**

5 Documentation for the various constraints available **13**

6 Indices and tables **15**

Python Module Index **17**

This package provides an easy interface to compute stability polygons.

One should create a StabilityPolygon by setting the robotMass, like so:

```
import stabilipy
poly = stabilipy.StabilityPolygon(57.5)
```

By default, a 3D robust static polyhedron is defined, see `stability.StabilityPolygon` for more information.

It is then necessary to create some contacts, and insert them in the polygon:

```
pos = [[[0., 0., 1.]], [[1., 0., 0.]]]
normals = [[[0., 0., 1.]], [[0.1, 0.1, 1.]]]
mu = 0.7

contacts = [stabilipy.Contact(mu, np.array(p).T,
                              stabilipy.utils.normalize(np.array(n).T))
            for p, n in zip(pos, normals)]
poly.contacts = contacts
```

Note that the normals *must* be of norm 1. You can now launch the computation:

```
poly.compute(stability.Mode.best, epsilon=1e-3, maxIter=50)
```

Compute takes many arguments, see `stability.StabilityPolygon.compute()` but the most importants are:

- Mode that determines if you want to reach a desired precision, iterate a number of times or best of both.
- epsilon sets the precision
- maxIter the number of iterations
- A number of plot_something keyword arguments are available.

Contents:

CHAPTER 1

Recursive Projection

1.1 Description of the algorithm

Recursive projection is an algorithm designed to compute the projection of a convex set. In general, it computes the approximation of a smooth set P. To do so, it generates converging polyhedral approximations of P, P_{inner} and P_{outer}. The algorithm only needs an "oracle", i.e. an algorithm or optimization problem that yields the extremal point of P in a given direction d. As a generic optimization problem:

$$\begin{aligned} max. \quad & d^T x \\ s.t. \quad & x \in P \end{aligned}$$

The solution x^* of this problem is an extremal point in the direction d. By solving repeatedly the above problem, we obtain:

- The convex hull of all x^* is P_{inner}.
- The intersection of all halfspaces $\{x \in \mathbb{R}^n | d^T x \leq d^T x^*\}$ forms P_{outer}

Now, the important point is how to choose the approriate sequence of directions d. To do so, we compute the *uncertainty volumes*, i.e. the cuts of P_{outer} by the supporting hyperplanes of P_{inner}. The direction d is chosen to be perpendicular to the supporting hyperplane that forms the largest uncertainty volume.

This is very useful to compute projections. Consider a convex body P in $\mathbb{R}^{n+m}$. Computing the projection of this body onto $\mathbb{R}^n$ can be done by specifying the following optimization problem:

$$\begin{aligned} max \quad & d^T x \\ s.t. \quad & (x, y) \in P \end{aligned}$$

This is particularly interesting when $m >> n$. In this case, computing the direct projection (see for example this page) is prohibitively expensive as the complexity is exponential in $m + n$. In our case, the complexity depends on the class of optimization problem being solved, but is typically polynomial in the dimension.

For more information, please refer to this paper.

1.2 How to use this class

This class is intended for developpers and researchers who wish to implement a new class of problems. If you are looking to compute stability or robust stability polygons and polyhedrons, please use *stabilipy.StabilityPolygon*. If you wish to compute the projection of a set of linear inequalities, please use *stabilipy.LinearProjection*. In general, one only needs to override the *stabilipy.RecursiveProjectionProblem.solve()* method.

Let us have a look at an example (available in *sphere.py*):

```
import stabilipy as stab


class SphereProjection(stab.RecursiveProjectionProblem):
  """Try to approximate a sphere of radius r"""

  def __init__(self, radius):
    """:param radius: Radius of the sphere
       :type radius: double"""
    stab.RecursiveProjectionProblem.__init__(self, dimension=3)
    self.radius = radius

  def solve(self, d):
    """We are computing a sphere so the extremal point in direction d is just r*d"""
    return self.radius*d

if __name__ == '__main__':
  sphere = SphereProjection(1.0)
  sphere.compute(solver='cdd', mode=stab.Mode.iteration, maxIter=50)
```

In this example we:

- Implement a class that extends RecursiveProjectionProblem
- Override solve to return the extremal point in the provided direction
- Instanciate that object and compute the approximation using *cdd* as our double-description package
- By default, this call will print the precision at each iteration and plot the result

See below for details of the API.

1.3 Class API

class stabilipy.**RecursiveProjectionProblem**(*dimension*, *verbosity=<Verbosity.info: 2>*)

Base class that encapsulates the recursive projection algorithm. To use it, you need to specify your problem by implementing the *solve* method. Then, this class will actually perform the projection.

Construct a projection problem.

Parameters

- **dimension** (*int*) – Dimension of the space on which you project.
- **verbosity** (*Verbosity*) – Verbosity of the output. Default to *info*.

clearAlgo()

Resets internal state

compute(*mode*, *maxIter=100*, *epsilon=0.0001*, *solver='cdd'*, *plot_error=False*, *plot_init=False*, *plot_step=False*, *plot_direction=False*, *record_anim=False*, *plot_final=True*, *fname_polys=None*)
Compute the polygon/polyhedron at a given precision or for a number of iterations or at best.

Parameters

- **mode** (stabilipy.Mode) – Stopping criterion.
- **maxIter** (*int*) – Maximum number of iterations.
- **epsilon** – Precision target.
- **solver** (stabilipy.backends) – Backend that will be used.
- **plot_error** – Make a running plot of the error during computation.
- **plot_init** – Plot the initial state of the algorithm.
- **plot_step** – Plot the state of the algorithm at each iteration.
- **plot_direction** – Plot the direction found for the next iteration.
- **record_anim** – Record all steps as images in files.
- **plot_final** – Plot the final polygon/polyhedron.
- **fname_polys** – Record successive iterations as text files.

make_problem()
This method is called upon launching the computation. Use it to build the complete problem from user-defined quantities. Does nothing by default.

outer_polyhedron()
Return the outer polyhedron as a set of vertices

plot()
Plot the current solution and polyhedrons

polyhedron()
Return the inner polyhedron as a set of vertices

save_outer(*fname*)
Save the outer polyhedron as a set of vertices :param fname: Filename to which the polyhedron is saved :type fname: string

save_polyhedron(*fname*)
Save the inner polyhedron as a set of vertices :param fname: Filename to which the polyhedron is saved :type fname: string

solve(*d*)
This method should return an extremal point in the given direction d, or None in case of error. You must reimplement this function to perform a computation or use one of the pre-implemented instances

Parameters **d** (*np.array((dim, 1))*) – Search direction

class stabilipy.**Mode**
All polygon computations should select a mode of operation.

- precision: will try to reach desired precision, however many iterations are required.
- iteration: will iterate for a number of iterations and stop, regardless of accuracy.
- best: will try to reach desired precision under the given number of iterations.

CHAPTER 2

Stability Polygon and contacts

class stabilipy.**StabilityPolygon**(*robotMass*, *dimension=3*, *gravity=-9.81*, *radius=2.0*, *force_lim=1.0*, *robust_sphere=-1*, *height=0.0*)

Algorithm to compute stability polygon according to Bretl et al. "Testing static equilibrium of legged robots". You need to first set some contacts and a robot mass Then call compute with desired precision. In 2D, computes a static stability polygon without discretizing cones. In 3D, computes a robust static stability polyhedron.

The default constructor to build a polygon/polyhedron.

Parameters

- **robotMass** – Mass of the robot
- **dimension** (*2, 3*) – Number of dimensions.
- **gravity** (*double*) – Intensity of gravity given along upwards z-axis.
- **radius** (*double*) – Radius of the CoM limitation constraint.
- **force_lim** (*double*) – Maximum force, expressed as a factor of the robot's weight.
- **robust_sphere** (*double*) – Robust radius to be used with spherical criterion. Negative disables
- **height** (*double*) – Height to be used when doing 2D robust

addCubeConstraint(*origin*, *length*)
Limit the CoM to | CoM - origin | < length

addDistConstraint(*origin*, *radius*)
Limit the CoM to || CoM - origin || < radius

addForceConstraint(*contacts*, *limit*)
Limit the sum of forces applied on contacts

addTorqueConstraint(*contacts*, *point*, *ub*, *lb=None*)
Add a limit on torque at a point over a set of contacts

clearConstraints()
Remove all constraints

make_problem()
Compute all matrices necessary to solving the problems. Only needs to be called once, because the problem shape never changes. This adds global dist constraint that should prevent CoM from going to infinity : ||com|| =< max. However, make sure you remove it between calls to compute or to set it to None when creating the polygon.

reset()
Remove all contacts, constraints and resets inner state

sample(*p*, *plot_final=True*, *plot_step=False*)
Test if a point is stable by iteratively refining the approximations

single_test(*p*)
Test if a single point is robust / non-robust without refining the approximations

CHAPTER 3

Linear Projection

3.1 Principle

Computing the projection of a convex set bounded by linear equalities and inequalities is a particular case of *Recursive Projection*. Indeed, in this case $x \in P$ can be directly written as:

$$Ax \leq b$$
$$Cx = d$$

And thus, finding extremal points amounts to solving Linear Programs (LP). Denoting the affine projection onto a smaller space by $y = Ex + f$ (same convention as Stéphane Caron), finding extremal points corresponding to a direction δ is done by solving:

$$\begin{aligned} max \quad & \delta^T(Ex + f) \\ s.t. \quad & Ax \leq b \\ & Cx = d \end{aligned}$$

A specific class is shown below.

3.2 Example of usage

The following example (contained in *hypercube.py*) shows how to project a 6D hypercube in 3D, resulting in a cube:

```
import stabilipy as stab
import numpy as np

if __name__ == '__main__':

    A = np.vstack((np.eye(6), -np.eye(6)))
    b = np.ones(12,)

    linear_proj = stab.LinearProjection(3, A, b, None, None)
    linear_proj.compute(stab.Mode.precision, solver='cdd', epsilon=1e-3)
```

Important notes:

- You need to specify the dimension you are projecting onto
- If you do not specify the projection operator E, f, it will default to projecting on the first *dimension* dimensions.

3.3 Class API

class `stabilipy.`**`LinearProjection`**(*dimension*, *A*, *b*, *C*, *d*, *E=None*, *f=None*)
Recursively compute the projection of a linear set: $Ax \leq b, Cx = d$ onto $y = Ex + f$

Create a linear projection problem.

Parameters

- **`dimension`** (*`int`*) – Dimension on which to project
- **`A`** (*`np.array(nrineq, dim)`*) – Linear inequality matrix
- **`b`** (*`np.array(nrineq,)`*) – Linear inequality RHS
- **`C`** (*`np.array(nreq, dim)`*) – Linear equality matrix
- **`d`** (*`np.array(nreq,)`*) – Linear equality RHS
- **`E`** (*`np.array(dimension, dim)`*) – Projection matrix
- **`f`** (*`np.array(dimension,)`*) – Projection offset

CHAPTER 4

Backends

These are the suitable backends for static stability polygon computation. Some backends are restricted to 2D/3D cases.

They all take as argument, a geometry engine. For now, only scipy is supported. Others are defined at least partially in geomengines.py:

- scipy: default and the only one supported as of now. We use its bindings of qhull.
- CGAL: Was supported but the available python bindings are too slow.
- Shapely: Does not support 3D properly
- Qhull-Sch: Custom bindings to qhullcpp for sch, that are not really usable as of now.

class stabilipy.backends.**CDDBackend**(*geomengine='scipy'*)
Using the CDD backend for polygon computation. This is the most polyvalent backend. Works on floating-point numbers. Requires pycddlib

Default constructor.

Parameters **geomengine** – underlying geometry engine. Only scipy is supported

class stabilipy.backends.**ParmaBackend**(*geomengine='scipy'*)
Backend using the Parma Polyhedra Library This is the most precise, and thus slow backend. Works on integer (unlimited precision through the use of GMP). Requires pyparma.

Default constructor.

Parameters **geomengine** – underlying geometry engine. Only scipy is supported

class stabilipy.backends.**PlainBackend**(*geomengine='scipy'*)
Plain Backend using the cdd backend for initialization. This is the simplest, fastest backend. However, only works on 2D polygons.

Default constructor.

Parameters **geomengine** – underlying geometry engine. Only scipy is supported.

class stabilipy.backends.**QhullBackend**(*geomengine='scipy'*)
Using the Qhull backend for polygon computation. This is an experimental backend that should yield better performance. Works on floating-point numbers. Requires scipy.

Default constructor.

Parameters **geomengine** – underlying geometry engine. Only scipy is supported

CHAPTER 5

Documentation for the various constraints available

class stabilipy.constraints.**Constraint**
Constraint types. Can only be inequality or conic.

class stabilipy.constraints.**CubeConstraint**(*origin*, *length*)
Constraint to limit position of the CoM to a cuboid centered at an origin

Default constructor. Origin will be clamped to the dimension of the polygon.

Parameters

- **origin** (*np.array(n, 1)*) – Origin of the cuboid.
- **length** – Length of the sides fo the box.

class stabilipy.constraints.**DistConstraint**(*origin*, *radius*)
Constraint to limit position of the CoM w.r. to an origin. The origin will be clamped to dimension of the polygon.

Default constructor.

Parameters

- **origin** (*np.array(n, 1)*) – Origin of the circle/sphere.
- **radius** – Radius of the circle/sphere

class stabilipy.constraints.**ForceConstraint**(*indexes*, *limit*)
Constraint to limit force applied on certain contacts

Default constructor.

Parameters

- **indexes** – Indexes of the contacts on which the constraint applies
- **limit** – Maximum force, expressed as percentage of robot weight

class stabilipy.constraints.**TorqueConstraint**(*indexes*, *point*, *ub*, *lb=None*)
Constraint to limit torque applied on certain contacts

Default constructor.

Parameters

- **indexes** – Indexes of the contacts on which the constraint applies
- **point** – Point where the torques are computed (3,1) array
- **ub** – Upper bound (3,1) array
- **lb** – Lower bound, can be None (3,1) array

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

stabilipy.backends, 11
stabilipy.constraints, 13

Index

A

addCubeConstraint() (stabilipy.StabilityPolygon method), 7
addDistConstraint() (stabilipy.StabilityPolygon method), 7
addForceConstraint() (stabilipy.StabilityPolygon method), 7
addTorqueConstraint() (stabilipy.StabilityPolygon method), 7

C

CDDBackend (class in stabilipy.backends), 11
clearAlgo() (stabilipy.RecursiveProjectionProblem method), 4
clearConstraints() (stabilipy.StabilityPolygon method), 7
compute() (stabilipy.RecursiveProjectionProblem method), 4
Constraint (class in stabilipy.constraints), 13
CubeConstraint (class in stabilipy.constraints), 13

D

DistConstraint (class in stabilipy.constraints), 13

F

ForceConstraint (class in stabilipy.constraints), 13

L

LinearProjection (class in stabilipy), 10

M

make_problem() (stabilipy.RecursiveProjectionProblem method), 5
make_problem() (stabilipy.StabilityPolygon method), 7

O

outer_polyhedron() (stabilipy.RecursiveProjectionProblem method), 5

P

ParmaBackend (class in stabilipy.backends), 11
PlainBackend (class in stabilipy.backends), 11
plot() (stabilipy.RecursiveProjectionProblem method), 5
polyhedron() (stabilipy.RecursiveProjectionProblem method), 5

Q

QhullBackend (class in stabilipy.backends), 11

R

RecursiveProjectionProblem (class in stabilipy), 4
reset() (stabilipy.StabilityPolygon method), 8

S

sample() (stabilipy.StabilityPolygon method), 8
save_outer() (stabilipy.RecursiveProjectionProblem method), 5
save_polyhedron() (stabilipy.RecursiveProjectionProblem method), 5
single_test() (stabilipy.StabilityPolygon method), 8
solve() (stabilipy.RecursiveProjectionProblem method), 5
stabilipy.backends (module), 11
stabilipy.constraints (module), 13
stabilipy.Mode (built-in class), 5
StabilityPolygon (class in stabilipy), 7

T

TorqueConstraint (class in stabilipy.constraints), 13