Dunshire Documentation

Release 0.1.1

Michael Orlitzky

Feb 10, 2023

CONTENTS

1 Overview **1**

2 Requirements **3**

3 User API Documentation **5**
- 3.1 dunshire.cones module . . . 5
- 3.2 dunshire.games module . . . 10

4 Background **27**
- 4.1 What is a linear game? . . . 27
- 4.2 Cone program formulation . . . 28

5 References **31**

6 Developer API Documentation **33**
- 6.1 dunshire.errors module . . . 33
- 6.2 dunshire.matrices module . . . 35
- 6.3 dunshire.options module . . . 41
- 6.4 test module . . . 42
- 6.5 test.matrices_test . . . 42
- 6.6 test.randomgen . . . 43
- 6.7 test.symmetric_linear_game_test module . . . 51

Bibliography **53**

Python Module Index **55**

CHAPTER

ONE

OVERVIEW

Dunshire is a library for solving linear games over symmetric cones. The notion of a symmetric linear (cone) game was introduced by Gowda and Ravindran [GowdaRav], and extended by Orlitzky to asymmetric cones with two interior points.

The state-of-the-art is that only symmetric games can be solved efficiently, and thus the linear games supported by Dunshire are a compromise between the two: the cones are symmetric, but the players get to choose two interior points.

In this game, we have two players who are competing for a "payoff." There is a symmetric cone K, a linear transformation L on the space in which K lives, and two points e_1 and e_2 in the interior of K. The players make their "moves" by choosing points from two strategy sets. Player one chooses an $\bar{x}$ from

$$\Delta_1 = \{x \in K \mid \langle x, e_2 \rangle = 1\}$$

and player two chooses a $\bar{y}$ from

$$\Delta_2 = \{y \in K \mid \langle y, e_1 \rangle = 1\}.$$

That ends the turn, and player one is paid $\langle L(\bar{x}), \bar{y} \rangle$ out of player two's pocket. As is usual in game theory, we suppose that player one wants to maximize his worst-case payoff, and that player two wants to minimize his worst-case *payout*. In other words, player one wants to solve the optimization problem,

$$\text{find} \max_{x \in \Delta_1} \min_{y \in \Delta_2} \langle L(x), y \rangle$$

while player two tries to (simultaneously) solve a similar problem,

$$\text{find} \min_{y \in \Delta_2} \max_{x \in \Delta_1} \langle L(x), y \rangle.$$

There is at least one pair $(\bar{x}, \bar{y})$ that solves these problems optimally, and Dunshire can find it. The optimal payoff, called *the value of the game*, is unique. At the moment, the symmetric cone K can be either the nonnegative orthant or the Lorentz "ice cream" cone in $\mathbb{R}^n$. Here are two of the simplest possible examples, showing off the ability to solve a game over both of those cones.

First, we use the nonnegative orthant in $\mathbb{R}^2$:

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(2)
>>> L = [[1,0],[0,1]]
>>> e1 = [1,1]
>>> e2 = e1
>>> G = SymmetricLinearGame(L,K,e1,e2)
>>> print(G.solution())
Game value: 0.500...
```

(continues on next page)

(continued from previous page)

```
Player 1 optimal:
  [0.500...]
  [0.500...]
Player 2 optimal:
  [0.500...]
  [0.500...]
```

Next we try the Lorentz ice-cream cone in $\mathbb{R}^2$:

```
>>> from dunshire import *
>>> K = IceCream(2)
>>> L = [[1,0],[0,1]]
>>> e1 = [1,0]
>>> e2 = e1
>>> G = SymmetricLinearGame(L,K,e1,e2)
>>> print(G.solution())
Game value: 1.000...
Player 1 optimal:
  [1.000...]
  [0.000...]
Player 2 optimal:
  [1.000...]
  [0.000...]
```

Note that these solutions are not unique, although the game values are.

CHAPTER

TWO

REQUIREMENTS

Dunshire is a Python library, so it needs a Python interpreter. To "build" it, you'll need Setuptools. The only other requirement is the CVXOPT library, available for most Linux distributions.

So, end users require:

- python (tested: 3.8, 3.9, 3.10)
- setuptools (tested: 65.7.0)
- cvxopt (tested: 1.3.0)

Developers will also want:

- GNU Make for convenience and to build the documentation.
- Pylint for code warnings.
- Sphinx to build the documentation.

CHAPTER

THREE

USER API DOCUMENTATION

You should only need to work with two modules, *dunshire.cones* and *dunshire.games*. For convenience, you can import everything from the `dunshire` package, and it will re-export what you need. For example,

```
from dunshire import *
K = IceCream(3)
L = [[1,-1,12],[0,1,22],[-17,1,0]]
e1 = [1,0.5,0.25]
e2 = [1,0.25,0.5]
G = SymmetricLinearGame(L,K,e1,e2)
G.solution()
```

3.1 dunshire.cones module

Class definitions for all of the symmetric cones (and their superclass, *SymmetricCone*) supported by CVXOPT.

class CartesianProduct(**factors*)

Bases: *SymmetricCone*

A cartesian product of symmetric cones, which is itself a symmetric cone.

Examples

```
>>> K = CartesianProduct(NonnegativeOrthant(3), IceCream(2))
>>> print(K)
Cartesian product of dimension 5 with 2 factors:
  * Nonnegative orthant in the real 3-space
  * Lorentz "ice cream" cone in the real 2-space
```

cvxopt_dims()

Return a dictionary of dimensions corresponding to the factors of this cartesian product. The format of this dictionary is described in the CVXOPT user's guide.

Returns
A dimension dictionary suitable to feed to CVXOPT.

Return type
dict

Examples

```
>>> K = CartesianProduct(NonnegativeOrthant(3),
...                      IceCream(2),
...                      IceCream(3))
>>> d = K.cvxopt_dims()
>>> (d['l'], d['q'], d['s'])
(3, [2, 3], [])
```

factors()

Return a tuple containing the factors (in order) of this cartesian product.

Returns
The factors of this cartesian product.

Return type
tuple of SymmetricCone.

Examples

```
>>> K = CartesianProduct(NonnegativeOrthant(3), IceCream(2))
>>> len(K.factors())
2
```

class IceCream(*dimension*)

Bases: *SymmetricCone*

The Lorentz "ice cream" cone in the given number of dimensions.

Examples

```
>>> K = IceCream(3)
>>> print(K)
Lorentz "ice cream" cone in the real 3-space
```

ball_radius(*point*)

Return the radius of a ball around `point` in this cone.

Since a radius cannot be negative, the `point` must be contained in this cone; otherwise, an error is raised.

The minimum distance from `point` to the complement of this cone will always occur at its projection onto that set. It is not hard to see that the projection is at a "down and out" angle of $\pi/4$ towards the outside of the cone. If one draws a right triangle involving the `point` and that projection, it becomes clear that the distance between `point` and its projection is a factor of $1/\sqrt{(2)}$ times the "horizontal" distance from `point` to boundary of this cone. For simplicity we take $1/2$ instead.

Parameters
point (*matrix*) – A point contained in this cone.

Returns
A radius `r` such that the ball of radius `r` centered at `point` is contained entirely within this cone.

Return type
float

Raises

- **TypeError** – If `point` is not a `cvxopt.base.matrix`.
- **TypeError** – If `point` has the wrong dimensions.
- **ValueError** – if `point` is not contained in this cone.

Examples

The height of `x` below is one (its first coordinate), and so the radius of the circle obtained from a height-one cross section is also one. Note that the last two coordinates of `x` are half of the way to the boundary of the cone, and in the direction of a 30-60-90 triangle. If one follows those coordinates, they hit at $\left(1, \frac{\sqrt{(3)}}{2}, \frac{1}{2}\right)$ having unit norm. Thus the "horizontal" distance to the boundary of the cone is $1 - \|x\|$, which simplifies to $1/2$. And rather than involve a square root, we divide by two for a final safe radius of $1/4$.

```
>>> from math import sqrt
>>> K = IceCream(3)
>>> x = matrix([1, sqrt(3)/4.0, 1/4.0])
>>> K.ball_radius(x)
0.25
```

class NonnegativeOrthant(*dimension*)

Bases: *SymmetricCone*

The nonnegative orthant in the given number of dimensions.

Examples

```
>>> K = NonnegativeOrthant(3)
>>> print(K)
Nonnegative orthant in the real 3-space
```

ball_radius(*point*)

Return the radius of a ball around `point` in this cone.

Since a radius cannot be negative, the `point` must be contained in this cone; otherwise, an error is raised.

The minimum distance from `point` to the complement of this cone will always occur at its projection onto that set. It is not hard to see that the projection is directly along one of the coordinates, and so the minimum distance from `point` to the boundary of this cone is the smallest coordinate of `point`. Therefore any radius less than that will work; we divide it in half somewhat arbitrarily.

Parameters
point (*matrix*) – A point contained in this cone.

Returns
A radius `r` such that the ball of radius `r` centered at `point` is contained entirely within this cone.

Return type
float

Raises

- **TypeError** – If `point` is not a `cvxopt.base.matrix`.

- **TypeError** – If `point` has the wrong dimensions.
- **ValueError** – if `point` is not contained in this cone.

Examples

```
>>> K = NonnegativeOrthant(5)
>>> K.ball_radius(matrix([1,2,3,4,5]))
0.5
```

class SymmetricCone(*dimension*)

Bases: `object`

An instance of a symmetric (self-dual and homogeneous) cone.

There are three types of symmetric cones supported by CVXOPT:

1. The nonnegative orthant in the real n-space.
2. The Lorentz "ice cream" cone, or the second-order cone.
3. The cone of symmetric positive-semidefinite matrices.

This class is intended to encompass them all.

When constructing a single symmetric cone (i.e. not a *CartesianProduct* of them), the only information that we need is its dimension. We take that dimension as a parameter, and store it for later.

Parameters
dimension (*int*) – The dimension of this cone.

Raises
ValueError – If you try to create a cone with dimension zero or less.

ball_radius(*point*)

Return the radius of a ball around `point` in this cone.

Since a radius cannot be negative, the `point` must be contained in this cone; otherwise, an error is raised.

Parameters
point (*matrix*) – A point contained in this cone.

Returns
A radius `r` such that the ball of radius `r` centered at `point` is contained entirely within this cone.

Return type
float

Raises
NotImplementedError – Always, this method must be implemented in subclasses.

Examples

```
>>> K = SymmetricCone(5)
>>> K.ball_radius(matrix([1,1,1,1,1]))
Traceback (most recent call last):
...
NotImplementedError
```

dimension()

Return the dimension of this symmetric cone.

The dimension of this symmetric cone is recorded during its creation. This method simply returns the recorded value, and should not need to be overridden in subclasses. We prefer to do any special computation in `__init__()` and record the result in `self._dimension`.

Returns
The stored dimension (from when this cone was constructed) of this cone.

Return type
int

Examples

```
>>> K = SymmetricCone(5)
>>> K.dimension()
5
```

class SymmetricPSD(*dimension*)

Bases: *SymmetricCone*

The cone of real symmetric positive-semidefinite matrices.

This cone has a dimension `n` associated with it, but we let `n` refer to the dimension of the domain of our matrices and not the dimension of the (much larger) space in which the matrices themselves live. In other words, our `n` is the `n` that appears in the usual notation S^n for symmetric matrices.

As a result, the cone `SymmetricPSD(n)` lives in a space of dimension $\left(n^2 + n\right)/2)$.

Examples

```
>>> K = SymmetricPSD(3)
>>> print(K)
Cone of symmetric positive-semidefinite matrices on the real 3-space
>>> K.dimension()
3
```

3.2 dunshire.games module

Symmetric linear games and their solutions.

This module contains the main *SymmetricLinearGame* class that knows how to solve a linear game.

class Solution(*game_value*, *p1_optimal*, *p2_optimal*)

Bases: `object`

A representation of the solution of a linear game. It should contain the value of the game, and both players' strategies.

Examples

```
>>> print(Solution(10, matrix([1,2]), matrix([3,4])))
Game value: 10.000...
Player 1 optimal:
  [ 1]
  [ 2]
Player 2 optimal:
  [ 3]
  [ 4]
```

game_value()

Return the game value for this solution.

Examples

```
>>> s = Solution(10, matrix([1,2]), matrix([3,4]))
>>> s.game_value()
10
```

player1_optimal()

Return player one's optimal strategy in this solution.

Examples

```
>>> s = Solution(10, matrix([1,2]), matrix([3,4]))
>>> print(s.player1_optimal())
[ 1]
[ 2]
```

player2_optimal()

Return player two's optimal strategy in this solution.

Examples

```
>>> s = Solution(10, matrix([1,2]), matrix([3,4]))
>>> print(s.player2_optimal())
[ 3]
[ 4]
```

class SymmetricLinearGame(*L*, *K*, *e1*, *e2*)

Bases: `object`

A representation of a symmetric linear game.

The data for a symmetric linear game are,

- A "payoff" operator `L`.
- A symmetric cone `K`.
- Two points `e1` and `e2` in the interior of `K`.

The ambient space is assumed to be the span of `K`.

With those data understood, the game is played as follows. Players one and two choose points x and y respectively, from their respective strategy sets,

$$\Delta_1 = \{x \in K \mid \langle x, e_2 \rangle = 1\}$$
$$\Delta_2 = \{y \in K \mid \langle y, e_1 \rangle = 1\}.$$

Afterwards, a "payout" is computed as $\langle L(x), y \rangle$ and is paid to player one out of player two's pocket. The game is therefore zero sum, and we suppose that player one would like to guarantee himself the largest minimum payout possible. That is, player one wishes to,

$$\text{maximize } \min_{y \in \Delta_2} (\langle L(x), y \rangle)$$
$$\text{subject to } x \in \Delta_1.$$

Player two has the simultaneous goal to,

$$\text{minimize } \max_{x \in \Delta_1} (\langle L(x), y \rangle)$$
$$\text{subject to } y \in \Delta_2.$$

These goals obviously conflict (the game is zero sum), but an existence theorem guarantees at least one optimal min-max solution from which neither player would like to deviate. This class is able to find such a solution.

Parameters

- **L** (*list of list of float*) – A matrix represented as a list of **rows**. This representation agrees with (for example) SageMath and NumPy, but not with CVXOPT (whose matrix constructor accepts a list of columns). In reality, `L` can be any iterable type of the correct length; however, you should be extremely wary of the way we interpret anything other than a list of rows.
- **K** (`dunshire.cones.SymmetricCone`) – The symmetric cone instance over which the game is played.
- **e1** (*iterable float*) – The interior point of `K` belonging to player one; it can be of any iterable type having the correct length.
- **e2** (*iterable float*) – The interior point of `K` belonging to player two; it can be of any enumerable type having the correct length.

Raises

ValueError – If either `e1` or `e2` lie outside of the cone `K`.

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,-5,-15],[-1,2,-3],[-12,-15,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,2,3]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG)
The linear game (L, K, e1, e2) where
  L = [  1  -5 -15]
      [ -1   2  -3]
      [-12 -15   1],
  K = Nonnegative orthant in the real 3-space,
  e1 = [ 1]
       [ 1]
       [ 1],
  e2 = [ 1]
       [ 2]
       [ 3]
```

Lists can (and probably should) be used for every argument:

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(2)
>>> L = [[1,0],[0,1]]
>>> e1 = [1,1]
>>> e2 = [1,1]
>>> G = SymmetricLinearGame(L, K, e1, e2)
>>> print(G)
The linear game (L, K, e1, e2) where
  L = [ 1  0]
      [ 0  1],
  K = Nonnegative orthant in the real 2-space,
  e1 = [ 1]
       [ 1],
  e2 = [ 1]
       [ 1]
```

The points `e1` and `e2` can also be passed as some other enumerable type (of the correct length) without much harm, since there is no row/column ambiguity:

```
>>> import cvxopt
>>> from dunshire import *
>>> K = NonnegativeOrthant(2)
>>> L = [[1,0],[0,1]]
>>> e1 = cvxopt.matrix([1,1])
>>> e2 = (1,1)
>>> G = SymmetricLinearGame(L, K, e1, e2)
>>> print(G)
The linear game (L, K, e1, e2) where
  L = [ 1  0]
      [ 0  1],
```

(continues on next page)

(continued from previous page)

```
  K = Nonnegative orthant in the real 2-space,
  e1 = [ 1]
       [ 1],
  e2 = [ 1]
       [ 1]
```

However, `L` will always be intepreted as a list of rows, even if it is passed as a `cvxopt.base.matrix` which is otherwise indexed by columns:

```
>>> import cvxopt
>>> from dunshire import *
>>> K = NonnegativeOrthant(2)
>>> L = [[1,2],[3,4]]
>>> e1 = [1,1]
>>> e2 = e1
>>> G = SymmetricLinearGame(L, K, e1, e2)
>>> print(G)
The linear game (L, K, e1, e2) where
  L = [ 1  2]
      [ 3  4],
  K = Nonnegative orthant in the real 2-space,
  e1 = [ 1]
       [ 1],
  e2 = [ 1]
       [ 1]
>>> L = cvxopt.matrix(L)
>>> print(L)
[ 1  3]
[ 2  4]

>>> G = SymmetricLinearGame(L, K, e1, e2)
>>> print(G)
The linear game (L, K, e1, e2) where
  L = [ 1  2]
      [ 3  4],
  K = Nonnegative orthant in the real 2-space,
  e1 = [ 1]
       [ 1],
  e2 = [ 1]
       [ 1]
```

A()

Return the matrix `A` used in our CVXOPT construction.

This matrix A appears on the right-hand side of $Ax = b$ in the statement of the CVXOPT conelp program.

Warning: It is not safe to cache any of the matrices passed to CVXOPT, because it can clobber them.

Returns

A 1-by-`(1 + self.dimension())` row vector. Its first entry is zero, and the rest are the entries of *e2()*.

Return type
matrix

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,1,1],[1,1,1],[1,1,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,2,3]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.A())
[0.0000000 1.0000000 2.0000000 3.0000000]
```

`C()`

Return the cone `C` used in our CVXOPT construction.

This is the cone over which the CVXOPT conelp program takes place.

Returns
The cartesian product of `K` with itself.

Return type
CartesianProduct

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[4,5,6],[7,8,9],[10,11,12]]
>>> e1 = [1,2,3]
>>> e2 = [1,1,1]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.C())
Cartesian product of dimension 6 with 2 factors:
  * Nonnegative orthant in the real 3-space
  * Nonnegative orthant in the real 3-space
```

`G()`

Return the matrix `G` used in our CVXOPT construction.

Thus matrix G appears on the left-hand side of $Gx + s = h$ in the statement of the CVXOPT conelp program.

Warning: It is not safe to cache any of the matrices passed to CVXOPT, because it can clobber them.

Returns
A `2*self.dimension()`-by-`(1 + self.dimension())` matrix.

Return type
matrix

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[4,5,6],[7,8,9],[10,11,12]]
>>> e1 = [1,2,3]
>>> e2 = [1,1,1]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.G())
[  0.0000000  -1.0000000   0.0000000   0.0000000]
[  0.0000000   0.0000000  -1.0000000   0.0000000]
[  0.0000000   0.0000000   0.0000000  -1.0000000]
[  1.0000000  -4.0000000  -5.0000000  -6.0000000]
[  2.0000000  -7.0000000  -8.0000000  -9.0000000]
[  3.0000000 -10.0000000 -11.0000000 -12.0000000]
```

K()

Return the cone over which this game is played.

Returns
The SymmetricCone over which this game is played.

Return type
SymmetricCone

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,-5,-15],[-1,2,-3],[-12,-15,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,2,3]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.K())
Nonnegative orthant in the real 3-space
```

L()

Return the matrix L passed to the constructor.

Returns
The matrix that defines this game's *payoff()* operator.

Return type
matrix

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,-5,-15],[-1,2,-3],[-12,-15,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,2,3]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.L())
[  1  -5 -15]
[ -1   2  -3]
[-12 -15   1]
```

static b()

Return the `b` vector used in our CVXOPT construction.

The vector b appears on the right-hand side of $Ax = b$ in the statement of the CVXOPT conelp program.

This method is static because the dimensions and entries of `b` are known beforehand, and don't depend on any other properties of the game.

Warning: It is not safe to cache any of the matrices passed to CVXOPT, because it can clobber them.

Returns
A 1-by-1 matrix containing a single entry 1.

Return type
matrix

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[4,5,6],[7,8,9],[10,11,12]]
>>> e1 = [1,2,3]
>>> e2 = [1,1,1]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.b())
[1.0000000]
```

c()

Return the vector `c` used in our CVXOPT construction.

The column vector c appears in the objective function value $\langle c, x \rangle$ in the statement of the CVXOPT conelp program.

Warning: It is not safe to cache any of the matrices passed to CVXOPT, because it can clobber them.

Returns
A *dimension()*-by-1 column vector.

Return type
matrix

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[4,5,6],[7,8,9],[10,11,12]]
>>> e1 = [1,2,3]
>>> e2 = [1,1,1]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.c())
[-1.0000000]
[ 0.0000000]
[ 0.0000000]
[ 0.0000000]
```

condition()

Return the condition number of this game.

In the CVXOPT construction of this game, two matrices `G` and `A` appear. When those matrices are nasty, numerical problems can show up. We define the condition number of this game to be the average of the condition numbers of `G` and `A` in the CVXOPT construction. If the condition number of this game is high, you can problems like `PoorScalingException`.

Random testing shows that a condition number of around `125` is about the best that we can solve reliably. However, the failures are intermittent, and you may get lucky with an ill-conditioned game.

Returns
A real number greater than or equal to one that measures how bad this game is numerically.

Return type
float

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(1)
>>> L = [[1]]
>>> e1 = [1]
>>> e2 = e1
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> SLG.condition()
1.809...
```

dimension()

Return the dimension of this game.

The dimension of a game is not needed for the theory, but it is useful for the implementation. We define the dimension of a game to be the dimension of its underlying cone. Or what is the same, the dimension of the space from which the strategies are chosen.

Returns
The dimension of the cone *K()*, or of the space where this game is played.

Return type
int

Examples

The dimension of a game over the nonnegative quadrant in the plane should be two (the dimension of the plane):

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(2)
>>> L = [[1,-5],[-1,2]]
>>> e1 = [1,1]
>>> e2 = [1,4]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> SLG.dimension()
2
```

dual()

Return the dual game to this game.

If $G = (L, K, e_1, e_2)$ is a linear game, then its dual is $G^* = (L^*, K^*, e_2, e_1)$. However, since this cone is symmetric, $K^* = K$.

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,-5,-15],[-1,2,-3],[-12,-15,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,2,3]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.dual())
The linear game (L, K, e1, e2) where
  L = [  1  -1 -12]
      [ -5   2 -15]
      [-15  -3   1],
  K = Nonnegative orthant in the real 3-space,
  e1 = [ 1]
       [ 2]
       [ 3],
  e2 = [ 1]
       [ 1]
       [ 1]
```

e1()

Return player one's interior point.

Returns
The point interior to *K()* affiliated with player one.

Return type
matrix

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,-5,-15],[-1,2,-3],[-12,-15,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,2,3]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.e1())
[ 1]
[ 1]
[ 1]
```

e2()

Return player two's interior point.

Returns
The point interior to *K()* affiliated with player one.

Return type
matrix

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,-5,-15],[-1,2,-3],[-12,-15,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,2,3]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.e2())
[ 1]
[ 2]
[ 3]
```

h()

Return the `h` vector used in our CVXOPT construction.

The h vector appears on the right-hand side of $Gx+s = h$ in the statement of the CVXOPT conelp program.

Warning: It is not safe to cache any of the matrices passed to CVXOPT, because it can clobber them.

Returns
A `2*self.dimension()`-by-1 column vector of zeros.

Return type
matrix

Examples

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[4,5,6],[7,8,9],[10,11,12]]
>>> e1 = [1,2,3]
>>> e2 = [1,1,1]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.h())
[0.0000000]
[0.0000000]
[0.0000000]
[0.0000000]
[0.0000000]
[0.0000000]
```

payoff(*strategy1*, *strategy2*)

Return the payoff associated with `strategy1` and `strategy2`.

The payoff operator takes pairs of strategies to a real number. For example, if player one's strategy is x and player two's strategy is y, then the associated payoff is $\langle L(x), y\rangle \in \mathbb{R}$. Here, L denotes the same linear operator as *L()*. This method computes the payoff given the two players' strategies.

Parameters

- **strategy1** (*matrix*) – Player one's strategy.
- **strategy2** (*matrix*) – Player two's strategy.

Returns

The payoff for the game when player one plays `strategy1` and player two plays `strategy2`.

Return type

float

Examples

The value of the game should be the payoff at the optimal strategies:

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,-5,-15],[-1,2,-3],[-12,-15,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,1,1]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> soln = SLG.solution()
>>> x_bar = soln.player1_optimal()
>>> y_bar = soln.player2_optimal()
>>> SLG.payoff(x_bar, y_bar) == soln.game_value()
True
```

player1_start()

Return a feasible starting point for player one.

This starting point is for the CVXOPT formulation and not for the original game. The basic premise is that if you scale *e2()* by the reciprocal of its squared norm, then you get a point in *K()* that makes a unit

inner product with *e2()*. We then get to choose the primal objective function value such that the constraint involving *L()* is satisfied.

Returns

A dictionary with two keys, `'x'` and `'s'`, which contain the vectors of the same name in the CVXOPT primal problem formulation.

The vector `x` consists of the primal objective function value concatenated with the strategy (for player one) that achieves it. The vector `s` is essentially a dummy variable, and is computed from the equality constraing in the CVXOPT primal problem.

Return type
dict

player2_start()

Return a feasible starting point for player two.

This starting point is for the CVXOPT formulation and not for the original game. The basic premise is that if you scale *e1()* by the reciprocal of its squared norm, then you get a point in *K()* that makes a unit inner product with *e1()*. We then get to choose the dual objective function value such that the constraint involving *L()* is satisfied.

Returns

A dictionary with two keys, `'y'` and `'z'`, which contain the vectors of the same name in the CVXOPT dual problem formulation.

The 1-by-1 vector `y` consists of the dual objective function value. The last *dimension()* entries of the vector `z` contain the strategy (for player two) that achieves it. The remaining entries of `z` are essentially dummy variables, computed from the equality constraint in the CVXOPT dual problem.

Return type
dict

solution()

Solve this linear game and return a *Solution*.

Returns
A *Solution* object describing the game's value and the optimal strategies of both players.

Return type
Solution

Raises

- ***GameUnsolvableException*** – If the game could not be solved (if an optimal solution to its associated cone program was not found).
- ***PoorScalingException*** – If the game could not be solved because CVXOPT crashed while trying to take the square root of a negative number.

Examples

This example is computed in Gowda and Ravindran in the section "The value of a Z-transformation":

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,-5,-15],[-1,2,-3],[-12,-15,1]]
>>> e1 = [1,1,1]
>>> e2 = [1,1,1]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.solution())
Game value: -6.172...
Player 1 optimal:
  [0.551...]
  [0.000...]
  [0.448...]
Player 2 optimal:
  [0.448...]
  [0.000...]
  [0.551...]
```

The value of the following game can be computed using the fact that the identity is invertible:

```
>>> from dunshire import *
>>> K = NonnegativeOrthant(3)
>>> L = [[1,0,0],[0,1,0],[0,0,1]]
>>> e1 = [1,2,3]
>>> e2 = [4,5,6]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.solution())
Game value: 0.031...
Player 1 optimal:
  [0.031...]
  [0.062...]
  [0.093...]
Player 2 optimal:
  [0.125...]
  [0.156...]
  [0.187...]
```

This is another Gowda/Ravindran example that is supposed to have a negative game value:

```
>>> from dunshire import *
>>> from dunshire.options import ABS_TOL
>>> L = [[1, -2], [-2, 1]]
>>> K = NonnegativeOrthant(2)
>>> e1 = [1, 1]
>>> e2 = e1
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> SLG.solution().game_value() < -ABS_TOL
True
```

The following two games are problematic numerically, but we should be able to solve them:

```
>>> from dunshire import *
>>> L = [[-0.95237953890954685221, 1.83474556206462535712],
...      [ 1.30481749924621448500, 1.65278664543326403447]]
>>> K = NonnegativeOrthant(2)
>>> e1 = [0.95477167524644313001, 0.63270781756540095397]
>>> e2 = [0.39633793037154141370, 0.10239281495640320530]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.solution())
Game value: 18.767...
Player 1 optimal:
  [0.000...]
  [9.766...]
Player 2 optimal:
  [1.047...]
  [0.000...]
```

```
>>> from dunshire import *
>>> L = [[1.54159395026049472754, 2.21344728574316684799],
...      [1.33147433507846657541, 1.17913616272988108769]]
>>> K = NonnegativeOrthant(2)
>>> e1 = [0.39903040089404784307, 0.12377403622479113410]
>>> e2 = [0.15695181142215544612, 0.85527381344651265405]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.solution())
Game value: 24.614...
Player 1 optimal:
  [6.371...]
  [0.000...]
Player 2 optimal:
  [2.506...]
  [0.000...]
```

This is another one that was difficult numerically, and caused trouble even after we fixed the first two:

```
>>> from dunshire import *
>>> L = [[57.222339086270523011 99, 41.706313734374603541 26],
...      [83.045125719850744872 02, 57.825818104069284686 37]]
>>> K = NonnegativeOrthant(2)
>>> e1 = [7.31887017043399268346, 0.89744171905822367474]
>>> e2 = [0.11099824781179848388, 6.12564670639315345113]
>>> SLG = SymmetricLinearGame(L,K,e1,e2)
>>> print(SLG.solution())
Game value: 70.437...
Player 1 optimal:
  [9.009...]
  [0.000...]
Player 2 optimal:
  [0.136...]
  [0.000...]
```

And finally, here's one that returns an "optimal" solution, but whose primal/dual objective function values are far apart:

```
>>> from dunshire import *
>>> L = [[ 6.49260076597376212248, -0.60528030227678542019],
...      [ 2.59896077096751731972, -0.97685530240286766457]]
>>> K = IceCream(2)
>>> e1 = [1, 0.43749513972645248661]
>>> e2 = [1, 0.46008379832200291260]
>>> SLG = SymmetricLinearGame(L, K, e1, e2)
>>> print(SLG.solution())
Game value: 11.596...
Player 1 optimal:
  [ 1.852...]
  [-1.852...]
Player 2 optimal:
  [ 1.777...]
  [-1.777...]
```

tolerance_scale(*solution*)

Return a scaling factor that should be applied to *dunshire.options.ABS_TOL* for this game.

When performing certain comparisons, the default tolerance *dunshire.options.ABS_TOL* may not be appropriate. For example, if we expect `x` and `y` to be within *dunshire.options.ABS_TOL* of each other, than the inner product of `L*x` and `y` can be as far apart as the spectral norm of `L` times the sum of the norms of `x` and `y`. Such a comparison is made in *solution()*, and in many of our unit tests.

The returned scaling factor found from the inner product mentioned above is

$$\|L\|_2 \left(\|\bar{x}\| + \|\bar{y}\|\right),$$

where $\bar{x}$ and $\bar{y}$ are optimal solutions for players one and two respectively. This scaling factor is not formally justified, but attempting anything smaller leads to test failures.

Warning: Optimal solutions are not unique, so the scaling factor obtained from `solution` may not work when comparing other solutions.

Parameters

solution (Solution) – A solution of this game, used to obtain the norms of the optimal strategies.

Returns

A scaling factor to be multiplied by *dunshire.options.ABS_TOL* when making comparisons involving solutions of this game.

Return type

float

Examples

The spectral norm of `L` in this case is around `5.464`, and the optimal strategies both have norm one, so we expect the tolerance scale to be somewhere around `2 * 5.464`, or `10.929`:

```
>>> from dunshire import *
>>> L = [[1,2],[3,4]]
>>> K = NonnegativeOrthant(2)
>>> e1 = [1,1]
>>> e2 = e1
>>> SLG = SymmetricLinearGame(L,K,e1,e2)
>>> SLG.tolerance_scale(SLG.solution())
10.929...
```

CHAPTER

FOUR

BACKGROUND

A linear game is a generalization of a two-person zero-sum matrix game [Karlin]. Classically, such a game involves a matrix $A \in \mathbb{R}^{n\times n}$ and a set (the unit simplex) of "strategies" $\Delta = \operatorname{conv}(\{e_1, e_2, \ldots, e_n\})$ from which two players are free to choose. If the players choose $x, y \in \Delta$ respectively, then the game is played by evaluating $y^T Ax$ as the "payoff" to the first player. His payoff comes from the second player, so that their total sums to zero.

Each player will try to maximize his payoff in this scenario, or—what is equivalent—try to minimize the payoff of his opponent. In fact, the existence of optimal strategies is guaranteed [Karlin] for both players. The *value* of the matrix game A is the payoff resulting from optimal play,

$$v(A) = \max_{x\in\Delta}\min_{y\in\Delta}\left(y^T Ax\right) = \min_{y\in\Delta}\max_{x\in\Delta}\left(y^T Ax\right).$$

The payoff to the first player in this case is $v(A)$. Corresponding to $v(A)$ is an optimal strategy pair $(\bar{x}, \bar{y}) \in \Delta \times \Delta$ such that

$$\bar{y}^T Ax \le v(A) = \bar{y}^T A\bar{x} \le y^T A\bar{x} \text{ for all } (x, y) \in \Delta \times \Delta.$$

The relationship between A, $\bar{x}$, $\bar{y}$, and $v(A)$ has been studied extensively [Kaplansky] [Raghavan]. Gowda and Ravindran [GowdaRav] were motivated by these results to ask if the matrix A can be replaced by a linear transformation L, and whether or not the unit simplex Δ can be replaced by a more general set—a base of a symmetric cone. In fact, one can go all the way to asymmetric (but still proper) cones. But, since Dunshire can only handle the symmetric case, we will pretend from now on that our cones need to be symmetric. This simplifies some definitions.

4.1 What is a linear game?

In the classical setting, the interpretation of the strategies as probabilities results in a strategy set $\Delta \subseteq \mathbb{R}^n_+$ that is compact, convex, and does not contain the origin. Moreover, any nonzero $x \in \mathbb{R}^n_+$ is a unique positive multiple $x = \lambda b$ of some $b \in \Delta$. Several existence and uniqueness results are predicated on those properties.

Definition. Suppose that K is a cone and $B \subseteq K$ does not contain the origin. If any nonzero $x \in K$ can be uniquely represented $x = \lambda b$ where $\lambda > 0$ and $b \in B$, then B is a *base* of K.

The set Δ is a compact convex base for the proper cone $\mathbb{R}^n_+$. Every $x \in \Delta$ also has entries that sum to unity, which can be abbreviated by the condition $\langle x, \mathbf{1}\rangle = 1$ where $\mathbf{1} = (1, 1, \ldots, 1)^T$ happens to lie in the interior of $\mathbb{R}^n_+$. In fact, the two conditions $x \in \mathbb{R}^n_+$ and $\langle x, \mathbf{1}\rangle = 1$ define Δ. This is no coincidence; whenever K is a symmetric cone and $e_2 \in \operatorname{int}(K)$, the set $\{x \in K \mid \langle x, e_2\rangle = 1\}$ is a compact convex base of K. This motivates a generalization where $\mathbb{R}^n_+$ is replaced by a symmetric cone.

Definition. Let V be a finite-dimensional real Hilbert space. A *linear game* in V is a tuple (L, K, e_1, e_2) where $L : V \to V$ is linear, the set K is a symmetric cone in V, and the points e_1 and e_2 belong to $\operatorname{int}(K)$.

Definition. The strategy sets for our linear game are

$$\Delta_1(L, K, e_1, e_2) = \{x \in K \mid \langle x, e_2\rangle = 1\}$$
$$\Delta_2(L, K, e_1, e_2) = \{y \in K \mid \langle y, e_1\rangle = 1\}.$$

Since $e_1, e_2 \in \operatorname{int}(K)$, these are bases for K. We will usually omit the arguments and write Δ_i to mean $\Delta_i(L, K, e_1, e_2)$.

To play the game (L, K, e_1, e_2), the first player chooses an $x \in \Delta_1$, and the second player independently chooses a $y \in \Delta_2$. This completes the turn, and the payoffs are determined by applying the payoff operator $(x, y) \mapsto \langle L(x), y\rangle$. The payoff to the first player is $\langle L(x), y\rangle$, and since we want the game to be zero-sum, the payoff to the second player is $-\langle L(x), y\rangle$.

The payoff operator is continuous in both arguments because it is bilinear and the ambient space is finite-dimensional. We constructed the strategy sets Δ_1 and Δ_2 to be compact and convex; as a result, Karlin's [Karlin] general min-max Theorem 1.5.1, guarantees the existence of optimal strategies for both players.

Definition. A pair $(\bar{x}, \bar{y}) \in \Delta_1 \times \Delta_2$ is an *optimal pair* for the game (L, K, e_1, e_2) if it satisfies the *saddle-point inequality*,

$$\langle L(x), \bar{y}\rangle \leq \langle L(\bar{x}), \bar{y}\rangle \leq \langle L(\bar{x}), y\rangle \text{ for all } (x, y) \in \Delta_1 \times \Delta_2.$$

At an optimal pair, neither player can unilaterally increase his payoff by changing his strategy. The value $\langle L(\bar{x}), \bar{y}\rangle$ is unique (by the same min-max theorem); it is shared by all optimal pairs. There exists at least one optimal pair $(\bar{x}, \bar{y})$ of the game (L, K, e_1, e_2) and its *value* is $v(L, K, e_1, e_2) = \langle L(\bar{x}), \bar{y}\rangle$.

Thanks to Karlin [Karlin], we have an equivalent characterization of a game's value that does not require us to have a particular optimal pair in mind,

$$v(L, K, e_1, e_2) = \max_{x\in\Delta_1} \min_{y\in\Delta_2} \langle L(x), y\rangle = \min_{y\in\Delta_2} \max_{x\in\Delta_1} \langle L(x), y\rangle.$$

Linear games reduce to two-person zero-sum matrix games in the right setting.

Example. If $K = \mathbb{R}^n_+$ in $V = \mathbb{R}^n$ and $e_1 = e_2 = (1, 1, \ldots, 1)^T \in \operatorname{int}(K)$, then $\Delta_1 = \Delta_2 = \Delta$. For any $L \in \mathbb{R}^{n\times n}$, the linear game (L, K, e_2, e_2) is a two-person zero-sum matrix game. Its payoff is $(x, y) \mapsto y^T Lx$, and its value is

$$v(L, K, e_1, e_2) = \max_{x\in\Delta} \min_{y\in\Delta} \left(y^T Lx\right) = \min_{y\in\Delta} \max_{x\in\Delta} \left(y^T Lx\right).$$

4.2 Cone program formulation

As early as 1947, von Neumann knew [Dantzig] that a two-person zero-sum matrix game could be expressed as a linear program. It turns out that the two concepts are equivalent, but turning a matrix game into a linear program is the simpler transformation. Cone programs are generalizations of linear programs where the cone is allowed to differ from the nonnegative orthant. We will see that any linear game can be formulated as a cone program, and if we're lucky, be solved.

Our main tool in formulating our cone program is the following theorem. It closely mimics a similar result in classical game theory where the cone is the nonnegative orthant and the result gives rise to a linear program. The symbols $\preccurlyeq$ and $\succcurlyeq$ indicate inequality with respect to the cone ordering; that is, $x \succcurlyeq y \iff x - y \in K$.

Theorem. In the game (L, K, e_1, e_2), we have $L^*(q) \preccurlyeq \nu e_2$ and $L(p) \succcurlyeq \nu e_1$ for $\nu \in \mathbb{R}$ if and only if ν is the value of the game (L, K, e_1, e_2) and (p, q) is optimal for it.

The proof of this theorem is not difficult, and the version for $e_1 \neq e_2$ can easily be deduced from the one given by Gowda and Ravidran [GowdaRav]. Let us now restate the objectives of the two players in terms of this theorem. Player one would like to,

$$\begin{aligned} \text{maximize} \quad & \nu \\ \text{subject to} \quad & p \in K \\ & \nu \in \mathbb{R} \\ & \langle p, e_2\rangle = 1 \\ & L(p) \succcurlyeq \nu e_1. \end{aligned}$$

Player two, on the other hand, would like to,

$$\begin{aligned} \text{minimize} \quad & \omega \\ \text{subject to} \quad & q \in K \\ & \omega \in \mathbb{R} \\ & \langle q, e_1 \rangle = 1 \\ & L^*(q) \preccurlyeq \omega e_2. \end{aligned}$$

The CVXOPT library can solve symmetric cone programs in the following primal/dual format:

$$\text{primal} = \begin{cases} \text{minimize } c^T x \\ \text{subject to } Gx + s = h \\ \qquad\qquad Ax = b \\ \qquad\qquad s \in C, \end{cases}$$

$$\text{dual} = \begin{cases} \text{maximize } -h^T z - b^T y \\ \text{subject to } G^T z + A^T y + c = 0 \\ \qquad\qquad z \in C. \end{cases}$$

We will now pull a rabbit out of the hat, and choose the matrices/vectors in these primal/dual programs so as to reconstruct the goals of the two players. Let,

$$\begin{aligned} C &= K \times K \\ x &= \begin{bmatrix} \nu \\ p \end{bmatrix} \\ y &= \begin{bmatrix} \omega \end{bmatrix} \\ b &= \begin{bmatrix} 1 \end{bmatrix} \\ h &= 0 \\ c &= \begin{bmatrix} -1 \\ 0 \end{bmatrix} \\ z &= \begin{bmatrix} z_1 \\ q \end{bmatrix} \\ A &= \begin{bmatrix} 0 & e_2^T \end{bmatrix} \\ G &= \begin{bmatrix} 0 & -I \\ e_1 & -L \end{bmatrix}. \end{aligned}$$

Substituting these values into the primal/dual CVXOPT cone programs recovers the objectives of player one (primal) and player two (dual) exactly. Therefore, we can use this formulation to solve a linear game, and that is precisely what Dunshire does.

CHAPTER

FIVE

REFERENCES

CHAPTER

SIX

DEVELOPER API DOCUMENTATION

This section contains API documentation that is of interest to developers, but not so much to end users. It documents our internal modules, exceptions, the test suite, and supporting code.

6.1 dunshire.errors module

Errors that can occur when solving a linear game.

exception GameUnsolvableException(*game*, *solution_dict*)

Bases: Exception

An exception raised when a game cannot be solved.

Every linear game has a solution. If we can't solve the conic program associated with a linear game, then something is wrong with either the model or the input, and this exception should be raised.

Parameters

- **game** (SymmetricLinearGame) – A copy of the game whose solution failed.
- **solution_dict** (*dict*) – The solution dictionary returned from the failed cone program.

Examples

```
>>> from dunshire import *
>>> K = IceCream(2)
>>> L = [[1,2],[3,4]]
>>> e1 = [1, 0.1]
>>> e2 = [3, 0.1]
>>> G = SymmetricLinearGame(L,K,e1,e2)
>>> d = {'residual as dual infeasibility certificate': None,
...      'y': matrix([1,1]),
...      'dual slack': 8.779496368228267e-10,
...      'z': matrix([1,1,0,0]),
...      's': None,
...      'primal infeasibility': None,
...      'status': 'primal infeasible',
...      'dual infeasibility': None,
...      'relative gap': None,
...      'iterations': 5,
...      'primal slack': None,
```

(continues on next page)

(continued from previous page)

```
...       'x': None,
...       'dual objective': 1.0,
...       'primal objective': None,
...       'gap': None,
...       'residual as primal infeasibility certificate':
...           3.986246886102996e-09}
>>> print(GameUnsolvableException(G,d))
Solution failed with result "primal infeasible."
The linear game (L, K, e1, e2) where
  L = [ 1  2]
      [ 3  4],
  K = Lorentz "ice cream" cone in the real 2-space,
  e1 = [1.0000000]
       [0.1000000],
  e2 = [3.0000000]
       [0.1000000]
CVXOPT returned:
  dual infeasibility: None
  dual objective: 1.0
  dual slack: 8.779496368228267e-10
  gap: None
  iterations: 5
  primal infeasibility: None
  primal objective: None
  primal slack: None
  relative gap: None
  residual as dual infeasibility certificate: None
  residual as primal infeasibility certificate: 3.986246886102996e-09
  s: None
  status: primal infeasible
  x: None
  y:
    [ 1]
    [ 1]
  z:
    [ 1]
    [ 1]
    [ 0]
    [ 0]
```

exception PoorScalingException(*game*)

Bases: `Exception`

An exception raised when poor scaling leads to solution errors.

Under certain circumstances, a problem that should be solvable can trigger errors in CVXOPT. The end result is the following `ValueError`:

```
Traceback (most recent call last):
...
  return math.sqrt(x[offset] - a) * math.sqrt(x[offset] + a)
ValueError: math domain error
```

This happens when one of the arguments to `math.sqrt()` is negative, but the underlying cause is elusive. We're

blaming it on "poor scaling," whatever that means.

Similar issues have been discussed a few times on the CVXOPT mailing list; for example,

1. https://groups.google.com/forum/#!msg/cvxopt/TeQGdc2b4Xc/j5_mQME_rvUJ
2. https://groups.google.com/forum/#!topic/cvxopt/HZrRfaoM0pk
3. https://groups.google.com/forum/#!topic/cvxopt/riFSxB31zU4

Parameters
game (SymmetricLinearGame) – A copy of the game whose solution failed.

Examples

```
>>> from dunshire import *
>>> K = IceCream(2)
>>> L = [[1,2],[3,4]]
>>> e1 = [1, 0.1]
>>> e2 = [3, 0.1]
>>> G = SymmetricLinearGame(L,K,e1,e2)
>>> print(PoorScalingException(G))
Solution failed due to poor scaling.
The linear game (L, K, e1, e2) where
  L = [ 1  2]
      [ 3  4],
  K = Lorentz "ice cream" cone in the real 2-space,
  e1 = [1.0000000]
       [0.1000000],
  e2 = [3.0000000]
       [0.1000000]
```

6.2 dunshire.matrices module

Utility functions for working with CVXOPT matrices (instances of the class:*cvxopt.base.matrix* class).

append_col(*left*, *right*)

Append two matrices side-by-side.

Parameters

- **left** (*matrix*) – The "original" matrix, the one that will wind up on the left.
- **right** (*matrix*) – The matrix to be appended on the right of `left`.

Returns
A new matrix consisting of `right` appended to the right of `left`.

Return type
matrix

Examples

```
>>> A = matrix([1,2,3,4], (2,2))
>>> B = matrix([5,6,7,8,9,10], (2,3))
>>> print(A)
[ 1  3]
[ 2  4]

>>> print(B)
[  5   7   9]
[  6   8  10]

>>> print(append_col(A,B))
[  1   3   5   7   9]
[  2   4   6   8  10]
```

append_row(*top*, *bottom*)

Append two matrices top-to-bottom.

Parameters

- **top** (*matrix*) – The "original" matrix, the one that will wind up on top.
- **bottom** (*matrix*) – The matrix to be appended below `top`.

Returns

A new matrix consisting of `bottom` appended below `top`.

Return type

matrix

Examples

```
>>> A = matrix([1,2,3,4], (2,2))
>>> B = matrix([5,6,7,8,9,10], (3,2))
>>> print(A)
[ 1  3]
[ 2  4]

>>> print(B)
[  5   8]
[  6   9]
[  7  10]

>>> print(append_row(A,B))
[  1   3]
[  2   4]
[  5   8]
[  6   9]
[  7  10]
```

condition_number(*mat*)

Return the condition number of the given matrix.

The condition number of a matrix quantifies how hard it is to do numerical computation with that matrix. It is usually defined as the ratio of the norm of the matrix to the norm of its inverse, and therefore depends on the norm used. One way to compute the condition number with respect to the 2-norm is as the ratio of the matrix's largest and smallest singular values. Since we have easy access to those singular values, that is the algorithm we use.

The larger the condition number is, the worse the matrix is.

Parameters
mat (*matrix*) – The matrix whose condition number you want.

Returns
The nonnegative condition number of `mat`.

Return type
float

Examples

```
>>> condition_number(identity(3))
1.0
```

```
>>> A = matrix([[2,1],[1,2]])
>>> abs(condition_number(A) - 3.0) < options.ABS_TOL
True
```

```
>>> A = matrix([[2,1j],[-1j,2]])
>>> abs(condition_number(A) - 3.0) < options.ABS_TOL
True
```

eigenvalues(*symmat*)

Return the eigenvalues of the given symmetric real matrix.

On the surface, this appears redundant to the *eigenvalues_re()* function. However, if we know in advance that our input is symmetric, a better algorithm can be used.

Parameters
symmat (*matrix*) – The real symmetric matrix whose eigenvalues you want.

Returns
A list of the eigenvalues (in no particular order) of `symmat`.

Return type
list of float

Raises
TypeError – If the input matrix is not symmetric.

Examples

```
>>> A = matrix([[2,1],[1,2]], tc='d')
>>> eigenvalues(A)
[1.0, 3.0]
```

If the input matrix is not symmetric, it may not have real eigenvalues, and we don't know what to do:

```
>>> A = matrix([[1,2],[3,4]])
>>> eigenvalues(A)
Traceback (most recent call last):
...
TypeError: input must be a symmetric real matrix
```

eigenvalues_re(*anymat*)

Return the real parts of the eigenvalues of the given square matrix.

Parameters
anymat (*matrix*) – The square matrix whose eigenvalues you want.

Returns
A list of the real parts (in no particular order) of the eigenvalues of anymat.

Return type
list of float

Raises
TypeError – If the input matrix is not square.

Examples

This is symmetric and has two real eigenvalues:

```
>>> A = matrix([[2,1],[1,2]], tc='d')
>>> sorted(eigenvalues_re(A))
[1.0, 3.0]
```

But this rotation matrix has eigenvalues *i* and *-i*, both of whose real parts are zero:

```
>>> A = matrix([[0,-1],[1,0]])
>>> eigenvalues_re(A)
[0.0, 0.0]
```

If the input matrix is not square, it doesn't have eigenvalues:

```
>>> A = matrix([[1,2],[3,4],[5,6]])
>>> eigenvalues_re(A)
Traceback (most recent call last):
...
TypeError: input matrix must be square
```

identity(*domain_dim*, *typecode='i'*)

Create an identity matrix of the given dimensions.

Parameters

- **domain_dim** (*int*) – The dimension of the vector space on which the identity will act.
- **typecode** (*{'i', 'd', 'z'}, optional*) – The type code for the returned matrix, defaults to 'i' for integers. Can also be 'd' for real double, or 'z' for complex double.

Returns
A domain_dim-by-domain_dim dense integer identity matrix.

Return type
matrix

Raises
ValueError – If you ask for the identity on zero or fewer dimensions.

Examples

```
>>> print(identity(3))
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
```

inner_product(*vec1*, *vec2*)

Compute the Euclidean inner product of two vectors.

Parameters

- **vec1** (*matrix*) – The first vector, whose inner product with vec2 you want.
- **vec2** (*matrix*) – The second vector, whose inner product with vec1 you want.

Returns
The inner product of vec1 and vec2.

Return type
float

Raises
TypeError – If the lengths of vec1 and vec2 differ.

Examples

```
>>> x = [1,2,3]
>>> y = [3,4,1]
>>> inner_product(x,y)
14
```

```
>>> x = matrix([1,1,1])
>>> y = matrix([2,3,4], (1,3))
>>> inner_product(x,y)
9
```

```
>>> x = [1,2,3]
>>> y = [1,1]
>>> inner_product(x,y)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: the lengths of vec1 and vec2 must match
```

norm(*matrix_or_vector*)

Return the Frobenius norm of a matrix or vector.

When the input is a vector, its matrix-Frobenius norm is the same thing as its vector-Euclidean norm.

Parameters
matrix_or_vector (*matrix*) – The matrix or vector whose norm you want.

Returns
The norm of `matrix_or_vector`.

Return type
float

Examples

```
>>> v = matrix([1,1])
>>> norm(v)
1.414...
```

```
>>> A = matrix([1,1,1,1], (2,2))
>>> norm(A)
2.0...
```

specnorm(*mat*)

Return the spectral norm of a matrix.

The spectral norm of a matrix is its largest singular value, and it corresponds to the operator norm induced by the vector Euclidean norm.

Parameters

- **mat** (*matrix*) – The matrix whose spectral norm you want.
- **Examples** –

```
>>> specnorm(identity(3))
1.0
```

```
>>> specnorm(5*identity(4))
5.0
```

vec(*mat*)

Create a long vector in column-major order from `mat`.

Parameters
mat (*matrix*) – Any sort of real matrix that you want written as a long vector.

Returns
An `len(mat)`-by-1 long column vector containign the entries of `mat` in column major order.

Return type
matrix

Examples

```
>>> A = matrix([[1,2],[3,4]])
>>> print(A)
[ 1  3]
[ 2  4]
```

```
>>> print(vec(A))
[ 1]
[ 2]
[ 3]
[ 4]
```

Note that if `mat` is a vector, this function is a no-op:

```
>>> v = matrix([1,2,3,4], (4,1))
>>> print(v)
[ 1]
[ 2]
[ 3]
[ 4]

>>> print(vec(v))
[ 1]
[ 2]
[ 3]
[ 4]
```

6.3 dunshire.options module

A place to collect the various options that "can be passed to the underlying engine." Just kidding, they're constants and you can't change them. But this makes the user interface real simple.

ABS_TOL = 1e-06

The absolute tolerance used in all "are these numbers equal" and "is this number less than (or equal to) that other number" tests. The CVXOPT default is `1e-7`, but loosening that a little reduces the number of "unknown" solutions that we get during random testing. Whether or not it improves the solubility of real problems is a question for the philosophers.

DEBUG_FLOAT_FORMAT = '%.20f'

The float output format to use when something goes wrong. If we need to reproduce a random test case, for example, then we need all of the digits of the things involved. If we try to recreate the problem using only, say, the first seven digits of each number, then the resulting game might not reproduce the failure.

FLOAT_FORMAT = '%.7f'

The default output format for floating point numbers.

6.4 test module

The whole test suite.

This module compiles the doctests and unittests from the rest of the codebase into one big TestSuite() and the runs it. It also provides a function *build_suite()* that merely builds the suite; the result can be used by setuptools.

build_suite(*doctests=True*)

Build our test suite, separately from running it.

Parameters
doctests (*bool*) – Do you want to build the doctests, too? During random testing, the answer may be "no."

run_suite(*suite*, *verbosity*)

Run all of the unit and doctests for the dunshire and *test* packages.

6.5 test.matrices_test

Unit tests for the functions in the *dunshire.matrices* module.

class AppendColTest(*methodName='runTest'*)

Bases: TestCase

Tests for the append_col() function.

test_new_dimensions()

If we append one matrix to another side-by-side, then the result should have the same number of rows as the two original matrices. However, the number of their columns should add up to the number of columns in the new combined matrix.

class AppendRowTest(*methodName='runTest'*)

Bases: TestCase

Tests for the *dunshire.matrices.append_row()* function.

test_new_dimensions()

If we append one matrix to another top-to-bottom, then the result should have the same number of columns as the two original matrices. However, the number of their rows should add up to the number of rows in the the new combined matrix.

class ConditionNumberTest(*methodName='runTest'*)

Bases: TestCase

Tests for the *dunshire.matrices.condition_number()* function.

test_condition_number_ge_one()

From the way that it is defined, the condition number should always be greater than or equal to one.

class EigenvaluesRealPartTest(*methodName='runTest'*)

Bases: TestCase

Tests for the *dunshire.matrices.eigenvalues_re()* function.

test_eigenvalues_re_input_not_clobbered()

The eigenvalue functions provided by CVXOPT/LAPACK like to overwrite the matrices that you pass into them as arguments. This test makes sure that our *dunshire.matrices.eigenvalues_re()* function does not do the same.

We use a deepcopy here in case the copy used in the *dunshire.matrices.eigenvalues_re()* function is insufficient. If copy didn't work and this test used it too, then this test would pass when it shouldn't.

test_eigenvalues_re_of_identity()

All eigenvalues of the identity matrix should be one.

class EigenvaluesTest(*methodName='runTest'*)

Bases: TestCase

Tests for the *dunshire.matrices.eigenvalues()* function.

test_eigenvalues_input_not_clobbered()

The eigenvalue functions provided by CVXOPT/LAPACK like to overwrite the matrices that you pass into them as arguments. This test makes sure that our eigenvalues() function does not do the same.

We use a deepcopy here in case the copy used in the eigenvalues() function is insufficient. If copy didn't work and this test used it too, then this test would pass when it shouldn't.

test_eigenvalues_of_identity()

All eigenvalues of the identity matrix should be one.

test_eigenvalues_of_symmat_are_real()

A real symmetric matrix has real eigenvalues, so if we start with a symmetric matrix, then the two functions *dunshire.matrices.eigenvalues()* and *dunshire.matrices.eigenvalues_re()* should agree on it.

class InnerProductTest(*methodName='runTest'*)

Bases: TestCase

Tests for the *dunshire.matrices.inner_product()* function.

test_inner_product_with_self_is_norm_squared()

Ensure that the func:*dunshire.matrices.inner_product* and *dunshire.matrices.norm()* functions are compatible by checking that the square of the norm of a vector is its inner product with itself.

class NormTest(*methodName='runTest'*)

Bases: TestCase

Tests for the *dunshire.matrices.norm()* function.

test_norm_is_nonnegative()

Test one of the properties of a norm, that it is nonnegative.

6.6 test.randomgen

Random thing generators used in the rest of the test suite.

MAX_COND = 125

The maximum condition number of a randomly-generated game. When the condition number of the games gets too high, we start to see PoorScalingException being thrown. There's no science to choosing the upper bound – it got lowered until those exceptions stopped popping up. It's at 125 because 129 doesn't work.

RANDOM_MAX = 10

When generating random real numbers or integers, this is used as the largest allowed magnitude. It keeps our condition numbers down and other properties within reason.

random_diagonal_matrix(*dims*)

Generate a random square matrix with zero off-diagonal entries.

These matrices are Lyapunov-like on the nonnegative orthant, as is fairly easy to see.

Parameters
dims (*int*) – The number of rows/columns you want in the returned matrix.

Returns
A new matrix whose diagonal entries are random floats chosen using *random_scalar()* and whose off-diagonal entries are zero.

Return type
matrix

Examples

```
>>> A = random_diagonal_matrix(3)
>>> A.size
(3, 3)
>>> A[0,1] == A[0,2] == A[1,0] == A[2,0] == A[1,2] == A[2,1] == 0
True
```

random_game()

Return a random game.

One of the functions,

1. *random_orthant_game()*
2. *random_icecream_game()*

is chosen at random and used to generate a random game.

Returns
A random game.

Return type
SymmetricLinearGame

Examples

```
>>> random_game()
<dunshire.games.SymmetricLinearGame object at 0x...>
```

random_icecream_game()

Generate a random game over the ice-cream cone.

We generate each of `L`, `K`, `e1`, and `e2` randomly within the constraints of the ice-cream cone, and then construct a game from them. The process is repeated until we generate a game with a condition number under *MAX_COND*.

Returns
A random game over some ice-cream cone.

Return type
SymmetricLinearGame

Examples

```
>>> random_icecream_game()
<dunshire.games.SymmetricLinearGame object at 0x...>
```

random_ll_game()

Return a random Lyapunov-like game.

One of the functions,

1. *random_ll_orthant_game()*
2. *random_ll_icecream_game()*

is chosen at random and used to generate a random game.

Returns
A random Lyapunov-like game.

Return type
SymmetricLinearGame

Examples

```
>>> random_ll_game()
<dunshire.games.SymmetricLinearGame object at 0x...>
```

random_ll_icecream_game()

Return a random Lyapunov game over some ice-cream cone.

We first construct a *random_icecream_game()* and then modify it to have a *random_lyapunov_like_icecream()* operator. That process is repeated until the condition number of the resulting game is within *MAX_COND*.

Returns
A random game over some ice-cream cone whose *dunshire.games.SymmetricLinearGame.payoff()* method is based on a Lyapunov-like *dunshire.games.SymmetricLinearGame.L()* operator.

Return type
SymmetricLinearGame

Examples

```
>>> random_ll_icecream_game()
<dunshire.games.SymmetricLinearGame object at 0x...>
```

random_ll_orthant_game()

Return a random Lyapunov game over some nonnegative orthant.

We first construct a *random_orthant_game()* and then modify it to have a *random_diagonal_matrix()* as its operator. Such things are Lyapunov-like on the nonnegative orthant. That process is repeated until the condition number of the resulting game is within *MAX_COND*.

Returns
A random game over some nonnegative orthant whose *dunshire.games.SymmetricLinearGame.payoff()* method is based on a Lyapunov-like *dunshire.games.SymmetricLinearGame.L()* operator.

Return type
SymmetricLinearGame

Examples

```
>>> random_ll_orthant_game()
<dunshire.games.SymmetricLinearGame object at 0x...>
```

random_lyapunov_like_icecream(*dims*)

Generate a random matrix Lyapunov-like on the ice-cream cone.

The form of these matrices is cited in Gowda and Tao [GowdaTao]. The scalar `a` and the vector `b` (using their notation) are easy to generate. The submatrix `D` is a little trickier, but it can be found noticing that $C + C^T = 0$ for a skew-symmetric matrix C implying that $C + C^T + (2a)\, I = (2a)\, I$. Thus we can stick an aI with each of C, C^T and let those be our D, D^T.

Parameters
dims (*int*) – The dimension of the ice-cream cone (not of the matrix you want!) on which the returned matrix should be Lyapunov-like.

Returns
A new matrix, Lyapunov-like on the ice-cream cone in `dims` dimensions, whose free entries are random floats chosen uniformly between negative and positive *RANDOM_MAX*.

Return type
matrix

References

Examples

```
>>> L = random_lyapunov_like_icecream(3)
>>> L.size
(3, 3)
```

```
>>> from dunshire.options import ABS_TOL
>>> from dunshire.matrices import inner_product
>>> x = matrix([1,1,0])
>>> s = matrix([1,-1,0])
>>> abs(inner_product(L*x, s)) < ABS_TOL
True
```

random_matrix(*row_count*, *column_count=None*)

Generate a random matrix.

Parameters

- **row_count** (*int*) – The number of rows you want in the returned matrix.

- **column_count** (*int*) – The number of columns you want in the returned matrix (default: the same as `row_count`).

Returns
A new matrix whose entries are random floats chosen uniformly between negative and positive *RANDOM_MAX*.

Return type
matrix

Examples

```
>>> A = random_matrix(3)
>>> A.size
(3, 3)
```

```
>>> A = random_matrix(3,2)
>>> A.size
(3, 2)
```

random_natural()

Generate a random natural number.

Returns
A random natural number between 1 and *RANDOM_MAX*, inclusive.

Return type
int

Examples

```
>>> 1 <= random_natural() <= RANDOM_MAX
True
```

random_nn_scalar()

Generate a random nonnegative scalar.

Returns
A random nonnegative real number between zero and *RANDOM_MAX*, inclusive.

Return type
float

Examples

```
>>> 0 <= random_nn_scalar() <= RANDOM_MAX
True
```

random_nn_scaling(*G*)

Scale the given game by a random nonnegative amount.

We re-attempt the scaling with a new random number until the resulting scaled game has an acceptable condition number.

Parameters
G (SymmetricLinearGame) – The game that you would like to scale.

Returns
A pair containing the both the scaling factor and the new scaled game.

Return type
(float, *SymmetricLinearGame*)

Examples

```
>>> from dunshire.matrices import norm
>>> from dunshire.options import ABS_TOL
>>> G = random_orthant_game()
>>> (alpha, H) = random_nn_scaling(G)
>>> alpha >= 0
True
>>> G.K() == H.K()
True
>>> norm(G.e1() - H.e1()) < ABS_TOL
True
>>> norm(G.e2() - H.e2()) < ABS_TOL
True
```

random_nonnegative_matrix(*row_count*, *column_count=None*)

Generate a random matrix with nonnegative entries.

Parameters

- **row_count** (*int*) – The number of rows you want in the returned matrix.
- **column_count** (*int*) – The number of columns you want in the returned matrix (default: the same as `row_count`).

Returns
A new matrix whose entries are chosen by *random_nn_scalar()*.

Return type
matrix

Examples

```
>>> A = random_nonnegative_matrix(3)
>>> A.size
(3, 3)
>>> all([entry >= 0 for entry in A])
True
```

```
>>> A = random_nonnegative_matrix(3,2)
>>> A.size
(3, 2)
>>> all([entry >= 0 for entry in A])
True
```

random_orthant_game()

Generate a random game over the nonnegative orthant.

We generate each of `L`, `K`, `e1`, and `e2` randomly within the constraints of the nonnegative orthant, and then construct a game from them. The process is repeated until we generate a game with a condition number under *MAX_COND*.

Returns
A random game over some nonnegative orthant.

Return type
SymmetricLinearGame

Examples

```
>>> random_orthant_game()
<dunshire.games.SymmetricLinearGame object at 0x...>
```

random_positive_orthant_game()

Return a random game over the nonnegative orthant with a positive operator.

We first construct a *random_orthant_game()* and then modify it to have a *random_nonnegative_matrix()* as its operator. That process is repeated until the condition number of the resulting game is within *MAX_COND*.

Returns
A random game over some nonnegative orthant whose *dunshire.games.SymmetricLinearGame.payoff()* method is based on a positive *dunshire.games.SymmetricLinearGame.L()* operator.

Return type
SymmetricLinearGame

Examples

```
>>> random_positive_orthant_game()
<dunshire.games.SymmetricLinearGame object at 0x...>
```

random_scalar()

Generate a random scalar.

Returns
A random real number between negative and positive *RANDOM_MAX*, inclusive.

Return type
float

Examples

```
>>> abs(random_scalar()) <= RANDOM_MAX
True
```

random_skew_symmetric_matrix(*dims*)

Generate a random skew-symmetrix matrix.

Parameters
dims (*int*) – The number of rows/columns you want in the returned matrix.

Returns
A new skew-matrix whose strictly above-diagonal entries are random floats chosen with *random_scalar()*.

Return type
matrix

Examples

```
>>> A = random_skew_symmetric_matrix(3)
>>> A.size
(3, 3)
```

```
>>> from dunshire.options import ABS_TOL
>>> from dunshire.matrices import norm
>>> A = random_skew_symmetric_matrix(random_natural())
>>> norm(A + A.trans()) < ABS_TOL
True
```

random_translation(*G*)

Translate the given game by a random amount.

We re-attempt the translation with new random scalars until the resulting translated game has an acceptable condition number.

Parameters
G (SymmetricLinearGame) – The game that you would like to translate.

Returns
A pair containing the both the translation distance and the new scaled game.

Return type
(float, *SymmetricLinearGame*)

Examples

```
>>> from dunshire.matrices import norm
>>> from dunshire.options import ABS_TOL
>>> G = random_orthant_game()
>>> (alpha, H) = random_translation(G)
>>> G.K() == H.K()
True
>>> norm(G.e1() - H.e1()) < ABS_TOL
```

(continues on next page)

(continued from previous page)

```
True
>>> norm(G.e2() - H.e2()) < ABS_TOL
True
```

6.7 test.symmetric_linear_game_test module

Unit tests for the SymmetricLinearGame class.

class SymmetricLinearGameTest(*methodName='runTest'*)

Bases: TestCase

Tests for the SymmetricLinearGame and Solution classes.

assert_lyapunov_works(*G*)

Check that Lyapunov games act the way we expect.

assert_opposite_game_works(*G*)

Check the value of the "opposite" game that gives rise to a value that is the negation of the original game. Comes from some corollary.

assert_orthogonality(*G*)

Two orthogonality relations hold at an optimal solution, and we check them here.

assert_player1_start_valid(*G*)

Ensure that player one's starting point satisfies both the equality and cone inequality in the CVXOPT primal problem.

assert_player2_start_valid(*G*)

Check that player two's starting point satisfies both the cone inequality in the CVXOPT dual problem.

assert_scaling_works(*G*)

Test that scaling L by a nonnegative number scales the value of the game by the same number.

assert_solutions_dont_change(*G*)

Solve G twice and check that the solutions agree.

assert_translation_works(*G*)

Check that translating L by alpha*(e1*e2.trans()) increases the value of the associated game by alpha.

assert_within_tol(*first*, *second*, *modifier=1*)

Test that first and second are equal within a multiple of our default tolerances.

Parameters

- **first** (*float*) – The first number to compare.
- **second** (*float*) – The second number to compare.
- **modifier** (*float*) – A scaling factor (default: 1) applied to the default tolerance for this comparison. If you have a poorly- conditioned matrix, for example, you may want to set this greater than one.

test_condition_lower_bound()

Ensure that the condition number of a game is greater than or equal to one.

It should be safe to compare these floats directly: we compute the condition number as the ratio of one nonnegative real number to a smaller nonnegative real number.

test_lyapunov_icecream()

Test that a Lyapunov game on the ice-cream cone works.

test_lyapunov_orthant()

Test that a Lyapunov game on the nonnegative orthant works.

test_opposite_game_icecream()

Like *test_opposite_game_orthant()*, except over the ice-cream cone.

test_opposite_game_orthant()

Test the value of the "opposite" game over the nonnegative orthant.

test_orthogonality_icecream()

Check the orthgonality relationships that hold for a solution over the ice-cream cone.

test_orthogonality_orthant()

Check the orthgonality relationships that hold for a solution over the nonnegative orthant.

test_player1_start_valid_icecream()

Ensure that player one's starting point is feasible over the ice-cream cone.

test_player1_start_valid_orthant()

Ensure that player one's starting point is feasible over the nonnegative orthant.

test_player2_start_valid_icecream()

Ensure that player two's starting point is feasible over the ice-cream cone.

test_player2_start_valid_orthant()

Ensure that player two's starting point is feasible over the nonnegative orthant.

test_positive_operator_value()

Test that a positive operator on the nonnegative orthant gives rise to a a game with a nonnegative value.

This test theoretically applies to the ice-cream cone as well, but we don't know how to make positive operators on that cone.

test_scaling_icecream()

The same test as `test_nonnegative_scaling_orthant()`, except over the ice cream cone.

test_scaling_orthant()

Test that scaling `L` by a nonnegative number scales the value of the game by the same number over the nonnegative orthant.

test_solutions_dont_change_icecream()

If we solve the same game twice over the ice-cream cone, then we should get the same solution both times. The solution to a game is not unique, but the process we use is (as far as we know) deterministic.

test_solutions_dont_change_orthant()

If we solve the same game twice over the nonnegative orthant, then we should get the same solution both times. The solution to a game is not unique, but the process we use is (as far as we know) deterministic.

test_translation_icecream()

The same as *test_translation_orthant()*, except over the ice cream cone.

test_translation_orthant()

Test that translation works over the nonnegative orthant.

BIBLIOGRAPHY

[Dantzig] G. B. Dantzig. Linear Programming and Extensions. Princeton University Press, Princeton, 1963.

[GowdaRav] M. S. Gowda and G. Ravindran. On the game-theoretic value of a linear transformation relative to a self-dual cone. Linear Algebra and its Applications, 469:440-463, 2015.

[Kaplansky] I. Kaplansky. A contribution to von Neumann's theory of games. Annals of Mathematics, 46(3):474-479, 1945.

[Karlin] S. Karlin. Mathematical Methods and Theory in Games, Programming, and Economics. Addison-Wesley Publishing Company, Reading, Massachusetts, 1959.

[Raghavan] T. Raghavan. Completely mixed games and M-matrices. Linear Algebra and its Applications, 21:35-45, 1978.

[GowdaTao] M. S. Gowda and J. Tao. On the bilinearity rank of a proper cone and Lyapunov-like transformations. Mathematical Programming, 147:155-170, 2014.

PYTHON MODULE INDEX

d

dunshire.cones, 5
dunshire.errors, 33
dunshire.games, 10
dunshire.matrices, 35
dunshire.options, 41

t

test, 42
test.matrices_test, 42
test.randomgen, 43
test.symmetric_linear_game_test, 51