

User Manual of the Packages Polyopt and PoolOpt

May 11, 2017

Chapter 1

Introduction

Polyopt is the implementation of the generalized BSOS and sparse-BSOS hierarchies [2]. PoolOpt is the package that uses Polyopt to solve pooling problems. These packages are written in Julia 0.5.

To use Polyopt and PoolOpt, you need to install two packages:

- MOSEK by using `julia> Pkg.add("MOSEK");`
- Combinatorics by using `julia> Pkg.add("Combinatorics");`

Also, you can install the packages using the following codes:

- `julia> Pkg.clone("https://github.com/MOSEK/Polyopt.jl.git");`
- `julia> Pkg.clone("https://github.com/AhmadrezaMarandi/PoolOpt.git");`

In the next chapters we define the available functions in the packages.

Chapter 2

Polyopt

As it was mentioned, Polyopt uses the generalization of the BSOS and sparse-BSOS hierarchy. The codes are based on the results in [2]. In this chapter, we present the available functions in this package.

2.1 `variable(string, int)`

To solve your problem, you first need to define the decision variables and constraints. This is possible by use of the function “variables”. This function gets two arguments as the input: a string and an integer. The string denotes the name of the variable, and the integer is the dimension of it.

Example 1 *julia> y=variables(“y”,5)*

5-element Array{Polyopt.PolyInt64,1}:

y1

y2

y3

y4

y5

Example 2

```
julia > x = variables("x", 4)
4 - element Array{PolyOpt.PolyInt64, 1} :
x1
x2
x3
x4
julia > f = x[1]^2 - x[2]^2 + x[3]^2 - x[4]^2 + x[1] - x[2]
- x4^2 + x3^2 - x2 - x2^2 + x1 + x1^2
julia > g = [2 * x[1]^2 + 3 * x[2]^2 + 2 * x[1] * x[2] + 2 * x[3]^2 + 3 * x[4]^2 + 2 * x[3] * x[4],
3 * x[1]^2 + 2 * x[2]^2 - 4 * x[1] * x[2] + 3 * x[3]^2 + 2 * x[4]^2 - 4 * x[3] * x[4],
x[1]^2 + 6 * x[2]^2 - 4 * x[1] * x[2] + x[3]^2 + 6 * x[4]^2 - 4 * x[3] * x[4],
x[1]^2 + 4 * x[2]^2 - 3 * x[1] * x[2] + x[3]^2 + 4 * x[4]^2 - 3 * x[3] * x[4],
2 * x[1]^2 + 5 * x[2]^2 + 3 * x[1] * x[2] + 2 * x[3]^2 + 5 * x[4]^2 + 3 * x[3] * x[4],
x[1], x[2], x[3], x[4]]
9 - element Array{PolyOpt.PolyInt64, 1} :
3 * x4^2 + 2 * x3 * x4 + 2 * x3^2 + 3 * x2^2 + 2 * x1 * x2 + 2 * x1^2
2 * x4^2 - 4 * x3 * x4 + 3 * x3^2 + 2 * x2^2 - 4 * x1 * x2 + 3 * x1^2
6 * x4^2 - 4 * x3 * x4 + x3^2 + 6 * x2^2 - 4 * x1 * x2 + x1^2
4 * x4^2 - 3 * x3 * x4 + x3^2 + 4 * x2^2 - 3 * x1 * x2 + x1^2
5 * x4^2 + 3 * x3 * x4 + 2 * x3^2 + 5 * x2^2 + 3 * x1 * x2 + 2 * x1^2
x1
x2
x3
x4
```

2.2 Polyopt.correlative_sparsity()

After constructing the problem, you can see the adjacent matrix of its graph using `Polyopt.correlative_sparsity()`.

Example 3

```
julia > Polyopt.correlative_sparsity(f, g)
4x4 sparse matrix with 16 Int64 nonzero entries :
 [1, 1] = 1
 [2, 1] = 1
 [3, 1] = 1
 [4, 1] = 1
 [1, 2] = 1
 [2, 2] = 1
 [3, 2] = 1
 [4, 2] = 1
 [1, 3] = 1
 [2, 3] = 1
 [3, 3] = 1
 [4, 3] = 1
 [1, 4] = 1
 [2, 4] = 1
 [3, 4] = 1
 [4, 4] = 1
```

2.3 Polyopt.chordal_embedding()

Using this function, you can find the sparsity pattern of the associated graph:

Example 4

```
julia > Polyopt.chordal_embedding(Polyopt.correlative_sparsity(f, g))
1 - element Array{Array{Int64, 1}, 1} :
 [1, 2, 3, 4]
```

2.4 bsosprob_chordal(d, k, I, f, g, eq)

This function will generate the level of the hierarchy. The arguments of this functions are:

d : level of the hierarchy;
k : the SOS polynomial in the hierarchy is with degree $2k$;
I : the sparsity pattern (maximal cliques of the graph associated to the problem)
or any appropriate pattern;
f : objective function;
g : inequality constraints;
eq : equality constraints.

2.5 solve_mosek(prob, tolrelgap)

This function solves the constructed level of the hierarchy using MOSEK.

prob : problem that is constructed for the level of the hierarchy;

tolrelgap : an optional argument to specify the preferred error gap.

Chapter 3

PoolOpt

This package solves pooling problems using Polyopt package. This package contains many well-known pooling problem instances such as Haverly1-3, Foulds1-4,..., and DeyGupte4 that is constructed in [1]. These instances can be reached by adding () after their name, like Haverly_1().

3.1 `elimination_equality(data)`

This function constructs the P-formulation of the pooling problem after elimination of equality constraints, which is introduced in [1]. “data” is the information of the pooling problem.

Example 5

```
julia > f, g = elimination_equality(Haverly_1());
```

Data was read.

```
===== Input – output variables are build! =====  
===== Pool – output variables are build! =====  
===== Concentration variables are build! =====  
===== Input – pool variables are build, considering elimination  
of the equality constraints! =====
```

```
===== The model is being constructed... =====
```

```
===== Objective function is constructed! =====
```

```
===== Constraints are constructed! =====
```

```
julia > f
```

```
– 1000.0 * y4 + 200.0 * y3 + 200.0000000000000142 * y2 – 2000.00000000000002 * y2 * y5 +  
1400.000000000000014 * y1 – 2000.00000000000002 * y1 * y5
```

```
julia > g
```

```
11 – elementArrayPolyopt.PolyFloat64, 1 :
```

```
1.0 – 2.0 * y3 – 2.0 * y1
```

```
1.0 – y4 – y2
```

```
0.05 * y3 + 0.150000000000000002 * y1 – 0.2 * y1 * y5
```

```
– 0.08333333333333333 * y4 + 0.08333333333333333 * y2 – 0.3333333333333333 * y2 * y5
```

```
y5
```

```
0.250000000000000001 * y2 * y5 + 0.250000000000000001 * y1 * y5
```

```
0.250000000000000001 * y2 – 0.250000000000000002 * y2 * y5 + 0.250000000000000001 * y1 – 0.250000000  
000000002 * y1 * y5
```

```
0.25 * y1
```

```
0.25 * y2
```

```
y3
```

```
y4
```


3.2 with_equality(data)

This function returns the P-formulation.

Example 6

```
julia> f, g, eq = with_equality(Haverly_1());
```

Data was read.

```
===== Input - output variables are build! =====
===== Pool - output variables are build! =====
===== Input - Pool variables are build! =====
===== Pool_specification variables are build! =====
===== The model is being constructed ... =====

===== Objective function is constructed! =====
```

```
julia> f
```

$3200.0 * y_7 + 1200.0 * y_6 - 1000.0 * y_4 + 200.0 * y_3 - 3000.0 * y_2 - 1800.0 * y_1$

```
julia> g
```

11 - element Array{PolyFloat64, 1}:

$1.0 - 2.0 * y_3 - 2.0 * y_1$

$1.0 - y_4 - y_2$

$0.05 * y_3 + 0.15000000000000002 * y_1 - 0.2 * y_1 * y_5$

$- 0.08333333333333333 * y_4 + 0.08333333333333333 * y_2 - 0.3333333333333333 * y_2 * y_5$

y_5

y_6

y_7

y_1

y_2

y_3

y_4

julia > eq

4 - elementArrayPolyopt.PolyFloat64, 1 :

0.3333333333333333 * y7 + 0.3333333333333333 * y6 - 0.3333333333333333 * y2 - 0.3333333333333333 * y1
 - 0.5 * y7 - 0.5 * y6 + 0.5 * y2 + 0.5 * y1

0.1111111111111111 * y7 + 0.3333333333333333 * y6 - 0.1111111111111111 * y2 - 0.2222222222222222 * y2 * y5 - 0.1111111111111111 * y1 - 0.2222222222222222 * y1 * y5
 - 0.1666666666666666 * y7 - 0.5 * y6 + 0.1666666666666666 * y2 + 0.3333333333333333 * y2 * y5 + 0.1666666666666666 * y1 + 0.3333333333333333 * y1 * y5

3.3 pooling_with_eq_BSOS(data, d, k)

This function solves the d th level of the BSOS hierarchy when the SOS polynomial in it is with degree $2k$.

Example 7

```
julia>pooling_with_eq_BSOS(Haverly_1(),1,1)
Data was read.
=====Input-output variables are build!=====
=====Pool-output variables are build!=====
=====Input-Pool variables are build!=====
=====Pool-specification variables are build!=====
=====The model is being constructed ... =====
=====Objective function is constructed! =====
3.014879 seconds (9.23 k allocations: 725.094 KB)
(16.)
>open file 'polyopt.task'
Problem
Name          :
Objective sense : max
Type          : CONIC (conic optimization problem)
Constraints    : 36
Cones         : 0
Scalar variables : 34
Matrix variables : 1
Integer variables : 0
Optimizer started.
conic_interior-point optimizer started.
```

```

Presolve started.
Linear dependency checker started.
Linear dependency checker terminated.
Eliminator - tries : 0 time :
0.00
Lin. dep. - tries : 1 time :
0.00
Lin. dep. - number : 0
Presolve terminated. Time: 0.00
Optimizer - threads : 4
Optimizer - solved problem : the primal
Optimizer - Constraints : 36
Optimizer - Cones : 1
Optimizer - Scalar variables : 31 conic :
4
Optimizer - Semi-definite variables: 1 scalarized :
36
Factor - setup time : 0.00 dense det. time :
0.00
Factor - ML order time : 0.00 GP order time :
0.00
Factor - nonzeros before factor : 666 after factor :
666
Factor - dense dim. : 0 flops :
2.18e+004
ITE PFEAS DFEAS GFEAS PRSTATUS POBJ DOBJ MU
TIME
0 1.9e+000 1.0e+000 1.0e+000 0.00e+000 0.000000000e+000 0.000000000e+000 1.
9e+000 0.00
1 6.1e-001 3.2e-001 6.1e-001 7.66e-001 -4.565784312e-001 -3.990256242e-001 3.
2e-001 0.00
2 2.7e-001 1.4e-001 4.6e-001 1.77e+000 -2.934437382e-001 -2.790044611e-001 1.
4e-001 0.00
3 1.4e-001 7.0e-002 2.9e-001 8.96e-001 -2.756713539e-001 -2.760321365e-001 7.
9e-002 0.00
4 3.7e-002 1.9e-002 1.5e-001 1.04e+000 -1.748599341e-001 -1.750857805e-001 1.
9e-002 0.00
5 7.9e-003 4.1e-003 6.9e-002 1.04e+000 -1.872951534e-001 -1.873214438e-001 4.
1e-003 0.00
6 1.0e-003 5.4e-004 2.6e-002 9.70e-001 -1.876969544e-001 -1.876622954e-001 5.
1e-004 0.00
7 3.6e-006 1.9e-006 1.5e-003 1.00e+000 -1.875031472e-001 -1.875030604e-001 1.
9e-006 0.00
8 2.3e-010 1.2e-010 1.2e-010 1.00e+000 -1.875000004e-001 -1.875000004e-001 1.
2e-010 0.01
Interior-point optimizer terminated. Time: 0.01.
Optimizer terminated. Time: 0.01

Interior-point solution summary
Problem status : PRIMAL_AND_DUAL_FEASIBLE
Solution status : OPTIMAL
Primal: obj: -1.8750000037e-001 nrm: 5e+000 viol. con: 1e-010 var: 2e-0
10 barvar: 0e+000
Dual: obj: -1.8750000036e-001 nrm: 1e+000 viol. con: 0e+000 var: 1e-0
10 barvar: 1e-010
(-600.0000011685997,"optimal",0.013156338)

```

Similarly, we have

- pooling_without_eq_BSOS(data, d, k)
- pooling_with_eq_Sparse_BSOS(data , d, k)
- pooling_without_eq_Sparse_BSOS(data , d, k)
- pooling_with_eq_Merge_Sparse_BSOS(data , d, k)
- pooling_without_eq_Merge_Sparse_BSOS(data , d, k)

where the last two function merge the cliques with high overlaps and then use sparse-BSOS hierarchy.

Bibliography

- [1] A. Marandi, J. Dahl, and E. de Klerk. A numerical evaluation of the bounded degree sum-of-squares hierarchy of lasserre, toh, and yang on the pooling problem. *Annals of Operations Research*, pages 1–26, 2017.
- [2] A. Marandi, E. De Klerk, and J. Dahl. Solving sparse polynomial optimization problems with chordal structure using the sparse, bounded-degree sum-of-squares hierarchy. *Optimization Online*, 2017.