Irene Documentation

Release 1.0.0

Mehdi Ghasemi

Dec 02, 2016

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

This is a brief documentation for using *Irene*. Irene was originally written to find reliable approximations for optimum value of an arbitrary optimization problem. It implements a modification of Lasserre's SDP Relaxations based on generalized truncated moment problem to handle general optimization problems algebraically.

Requirements and dependencies

This is a python package, so clearly python is an obvious requirement. Irene relies on the following packages:

- **for vector calculations:**
  - NumPy.
- **for symbolic computations:**
  - SymPy.
- **for semidefinite optimization, at least one of the following is required:**
  - cvxopt,
  - dsdp,
  - sdpa,
  - csdp.

Download

Irene can be obtained from https://github.com/mghasemi/Irene.

Installation

To install *Irene*, run the following in terminal:

```
sudo python setup.py install
```

Documentation

The documentation of *Irene* is prepared via sphinx.

To compile html version of the documentation run:

```
$Irene/doc/make html
```

To make a pdf file,subject to existence of `latexpdf` run:

```
$Irene/doc/make latexpdf
```

License

Irene is distributed under MIT license:

MIT License

CHAPTER

TWO

SEMIDEFINITE PROGRAMMING

A *positive semidefinite* matrix is a symmetric real matrix whose eigenvalues are all nonnegative. A semidefinite programming problem is simply a linear program where the solutions are positive semidefinite matrices instead of points in Euclidean space.

Primal and Dual formulations

A typical semidefinite program (SDP for short) in the primal form is the following optimization problem:

$$\begin{cases} \min & \sum_{i=1}^{m} b_i x_i \\ \text{subject to} & \\ & \sum_{i=1}^{m} A_{ij} x_i - C_j \succeq 0 \quad j = 1, \dots, k. \end{cases}$$

The dual program associated to the above SDP will be the following:

$$\begin{cases} \max & \sum_{j=1}^{k} tr(C_j \times Z_j) \\ \text{subject to} & \\ & \sum_{j=1}^{k} tr(A_{ij} \times Z_j) = b_i \quad i = 1, \dots, m, \\ & Z_j \succeq 0 \qquad\qquad\qquad\quad j = 1, \dots, k. \end{cases}$$

For convenience, we use a block representation for the matrices as follows:

$$C = \begin{pmatrix} C_1 & 0 & 0 & \dots \\ 0 & C_2 & 0 & \dots \\ \vdots & \dots & \ddots & \vdots \\ 0 & \dots & 0 & C_k \end{pmatrix},$$

and

$$A_i = \begin{pmatrix} A_{i1} & 0 & 0 & \dots \\ 0 & A_{i2} & 0 & \dots \\ \vdots & \dots & \ddots & \vdots \\ 0 & \dots & 0 & A_{ik} \end{pmatrix}.$$

This simplifies the k constraints of the primal form in to one constraint $\sum_{i=1}^{m} A_i x_i - C \succeq 0$ and the objective and constraints of the dual form as $tr(C \times Y)$ and $tr(A_i \times Z_i) = b_i$ for $i = 1, dots, m$.

The `sdp` class

The `sdp` class provides an interface to solve semidefinite programs using various range of well-known SDP solvers. Currently, the following solvers are supported:

CVXOPT

This is a python native convex optimization solver which can be obtained from CVXOPT. Beside semidefinite programs, it has various other solvers to handle convex optimization problems. In order to use this solver, the python package `CVXOPT` must be installed.

DSDP

If DSDP and `CVXOPT` are installed and `DSDP` is callable from command line, then it can be used as a SDP solver. Note that the current implementation uses `CVXOPT` to call `DSDP`, so `CVXOPT` is a requirement too.

SDPA

In case one manages to install SDPA and it can be called from command line, one can use `SDPA` as a SDP solver.

CSDP

Also, if csdp is installed and can be reached from command, then it can be used to solve SDP problems through `sdp` class.

To initialize and set the solver to one of the above simply use:

```
SDP = sdp('cvxopt') # initializes and uses `cvxopt` as solver.
```

Set the b vector:

To set the vector $b = (b_1, \ldots, b_m)$ one should use the method *sdp.SetObjective* which takes a list or a numpy array of numbers as b.

Set a block constraint:

To introduce the block of matrices $A_{i1}, \ldots, A_{ik}$ associated with x_i, one should use the method *sdp.AddConstraintBlock* that takes a list of matrices as blocks.

Set the constant block *C*:

The method *sdp.AddConstantBlock* takes a list of square matrices and use them to construct C.

Solve the input SDP:

To solve the input SDP simply call the method *sdp.solve()*. This will call the selected solver on the entered SDP and the output of the solver will be set as dictionary in *sdp.Info* with the following keys:

- *PObj*: The value of the primal objective.
- *DObj*: The value of the dual objective.
- *X*: The final X matrix.
- *Z*: The final Z matrix.

- *Status*: The final status of the solver.
- *CPU*: Total run time of the solver.

Example:

Consider the following SDP:

$$\begin{cases} \min & x_1 - x_2 + x_3 \\ \text{subject to} & \begin{pmatrix} 7 & 11 \\ 11 & -3 \end{pmatrix} x_1 + \begin{pmatrix} -7 & 18 \\ 18 & -8 \end{pmatrix} x_2 + \begin{pmatrix} 2 & 8 \\ 8 & -1 \end{pmatrix} x_3 \succeq \begin{pmatrix} -33 & 9 \\ 9 & -26 \end{pmatrix} \\ & \begin{pmatrix} 21 & 11 & 0 \\ 11 & -10 & -8 \\ 0 & -8 & -5 \end{pmatrix} x_1 + \begin{pmatrix} 0 & -10 & -16 \\ -10 & 10 & 10 \\ -16 & 10 & -3 \end{pmatrix} x_2 + \begin{pmatrix} 5 & -2 & 17 \\ -2 & 6 & -8 \\ 17 & -8 & -6 \end{pmatrix} x_3 \succeq \begin{pmatrix} -14 & -9 & -40 \\ -9 & -91 & -10 \\ -40 & -10 & -15 \end{pmatrix} \end{cases}$$

The following code solves the above program:

```
from numpy import matrix
from Irene import sdp
b = [1, -1, 1]
C = [matrix([[-33, 9], [9, -26]]),
     matrix([[-14, -9, -40], [-9, -91, -10], [-40, -10, -15]])]
A1 = [matrix([[7, 11], [11, -3]]),
      matrix([[21, 11, 0], [11, -10, -8], [0, -8, -5]])]
A2 = [matrix([[-7, 18], [18, -8]]),
      matrix([[0, -10, -16], [-10, 10, 10], [-16, 10, -3]])]
A3 = [matrix([[2, 8], [8, -1]]),
      matrix([[5, -2, 17], [-2, 6, -8], [17, -8, -6]])]
SDP = sdp('cvxopt')
SDP.SetObjective(b)
SDP.AddConstantBlock(C)
SDP.AddConstraintBlock(A1)
SDP.AddConstraintBlock(A2)
SDP.AddConstraintBlock(A3)
SDP.solve()
print SDP.Info
```

CHAPTER

THREE

OPTIMIZATION

Let X be a nonempty topological space and A be a unital sub-algebra of continuous functions over X which separates points of X. We consider the following optimization problem:

$$\begin{cases} \min & f(x) \\ \text{subject to} & \\ & g_i(x) \geq 0 \quad i = 1, \ldots, m. \end{cases}$$

Denote the feasibility set of the above program by K (i.e., $K = \{x \in X : g_i(x) \geq 0,\ i = 1, \ldots, m\}$). Let ρ be the optimum value of the above program and $\mathcal{M}_1^+(K)$ be the space of all probability Borel measures supported on K. One can show that:

$$\rho = \inf_{\mu \in \mathcal{M}_1^+(K)} \int f \, d\mu.$$

This associates a K-positive linear functional L_μ to every measure $\mu \in \mathcal{M}_1^+(K)$. Let us denote the set of all elements of A nonnegative on K by $Psd_A(K)$. If $\exists p \in Psd_A(K)$ such that $p^{-1}([0, n])$ is compact for each $n \in \mathbb{N}$, then one can show that every K-positive linear functional admits an integral representation via a Borel measure on K (Marshall's generalization of Haviland's theorem). Let $Q_{\mathbf{g}}$ be the quadratic module generated by $g_1, \ldots, g_m$, i.e, the set of all elements in A of the form

$$\sigma_0 + \sigma_1 g_1 + \cdots + \sigma_m g_m, \tag{3.1}$$

where $\sigma_0, \ldots, \sigma_m \in \sum A^2$ are sums of squares of elements of A. A quadratic module Q is said to be Archimedean if for every $h \in A$ there exists $M > 0$ such that $M \pm h \in Q$. By Jacobi's representation theorem, if Q is Archimedean and $h > 0$ on K, where $K = \{x \in X : g(x) \geq 0 \ \forall g \in Q\}$, then $h \in Q$. Since Q is Archimedean, K is compact and this implies that if a linear functional on A is nonnegative on Q, then it is K-positive and hence admits an integral representation. Therefore:

$$\rho = \inf_{\substack{L(Q) \geq 0 \\ L(1) = 1}} L(f).$$

Let $Q = Q_{\mathbf{g}}$ and $L(Q) \subseteq [0, \infty)$. Then clearly $L(\sum A^2) \subseteq [0, \infty)$ which means L is positive semidefinite. Moreover, for each $i = 1, \ldots, m$, $L(g_i \sum A^2) \subseteq [0, \infty)$ which means the maps

$$\begin{array}{rccc} L_{g_i} : A & \longrightarrow & \mathbb{R} \\ h & \mapsto & L(g_i h) \end{array}$$

are positive semidefinite. So the optimum value of the following program is still equal to ρ:

$$\begin{cases} \min & L(f) \\ \text{subject to} & \\ & L \succeq 0 \\ & L_{g_i} \succeq 0 \quad i = 1, \ldots, m. \end{cases} \tag{3.2}$$

This, still is not a semidefinite program as each constraint is infinite dimensional. One plausible idea is to consider functionals on finite dimensional subspaces of A containing $f, g_1, \ldots, g_m$. This was done by Lasserre for polynomials *[JBL]*.

Let $B \subseteq A$ be a linear subspace. If $L : A \longrightarrow \mathbb{R}$ is K-positive, so is its restriction on B. But generally, K-positive maps on B do not extend to K-positive one on A and hence existence of integral representations are not guaranteed. Under a rather mild condition, this issue can be resolved:

Theorem. *[GIKM]* Let $K \subseteq X$ be compact, $B \subseteq A$ a linear subspace such that there exists $p \in B$ strictly positive on K. Then every linear functional $L : B \longrightarrow \mathbb{R}$ satisfying $L(Psd_B(K)) \subseteq [0, \infty)$ admits an integral representation via a Borel measure supported on K.

Now taking B to be a finite dimensional linear space containing $f, g_1, \ldots, g_m$ and satisfying the assumptions of the above theorem, turns (3.2) into a semidefinite program. Note that this does not imply that the optimum value of the resulting SDP is equal to ρ since

- $Q_{\mathbf{g}} \cap B \neq Psd_B(K)$ and,
- there may not exist a decomposition of $f - \rho$ as in (3.1) inside B (i.e., the summands may not belong to B).

Thus, the optimum value just gives a lower bound for ρ. But walking through a K-frame, as explained in *[GIKM]* constructs a net of lower bounds for ρ which approaches ρ, eventually.

I practice, one only needs to find a sufficiently big finite dimensional linear space which contains $f, g_1, \ldots, g_m$ and a (3.1) decomposition of $f - \rho$ can be found within that space. Therefore, the convergence happens in finitely many steps, subject to finding a suitable K-frame for the problem.

The significance of this method is that it converts any optimization problem into finitely many semidefinite programs whose optimum values approaches the optimum value of the original program and semidefinite programs can be solved in polynomial time. Although, this suggests that the NP-complete problem of optimization can be solved in P-time, but since the number of SDPs that is required to reach the optimum is unknown and such a bound does not exists when dealing with Archimedean modules.

Polynomial Optimization

The SDP relaxation method was originally introduced by Lasserre *[JBL]* for polynomial optimization problem and excellent software packages such as GloptiPoly and ncpol2sdpa exist to handle constraint polynomial optimization problems.

Irene uses sympy for symbolic computations, so, it always need to be imported and the symbolic variables must be introduced. Once these steps are done, the objective and constraints should be entered using `SetObjective` and *AddConstraint* methods. the method `MomentsOrd` takes the relaxation degree upon user's request, otherwise the minimum relaxation degree will be used. The default SDP solver is `CVXOPT` which can be modified via `SetSDPSolver` method. Currently `CVXOPT`, `DSDP`, `SDPA` and `CSDP` are supported. Next step is initialization of the SDP by `InitSDP` and finally solving the SDP via `Minimize` and the output will be stored in the `Solution` variable as a python dictionary.

Example Solve the following polynomial optimization problem:

$$\begin{cases} \min & -2x + y - z \\ \text{subject to} & 24 - 20x + 9y - 13z + 4x^2 - 4xy \\ & +4xz + 2y^2 - 2yz + 2z^2 \geq 0 \\ & x + y + z \leq 4 \\ & 3y + z \leq 6 \\ & 0 \leq x \leq 2 \\ & y \geq 0 \\ & 0 \leq z \leq 3. \end{cases}$$

The following program uses relaxation of degree 3 and *sdpa* to solve the above problem:

```
from sympy import *
from Irene import *
# introduce variables
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
# initiate the Relaxation object
Rlx = SDPRelaxations([x, y, z])
# set the objective
Rlx.SetObjective(-2 * x + y - z)
# add support constraints
Rlx.AddConstraint(24 - 20 * x + 9 * y - 13 * z + 4 * x**2 -
                  4 * x * y + 4 * x * z + 2 * y**2 - 2 * y * z + 2 * z**2 >= 0)
Rlx.AddConstraint(x + y + z <= 4)
Rlx.AddConstraint(3 * y + z <= 6)
Rlx.AddConstraint(x >= 0)
Rlx.AddConstraint(x <= 2)
Rlx.AddConstraint(y >= 0)
Rlx.AddConstraint(z >= 0)
Rlx.AddConstraint(z <= 3)
# set the relaxation order
Rlx.MomentsOrd(3)
# set the solver
Rlx.SetSDPSolver('dsdp')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
# output
print Rlx.Solution
```

The output looks like:

```
Solution of a Semidefinite Program:
                Solver: DSDP
                Status: Optimal
   Initialization Time: 31.0824120045 seconds
              Run Time: 1.056733 seconds
Primal Objective Value: -4.06848294478
  Dual Objective Value: -4.06848289445
Feasible solution for moments of order 3
```

Moment Constraints

Initially the only constraints forced on the moments are those in (3.2). We can also force user defined constraints on the moments by calling `MomentConstraint` on a `Mom` object. The following adds two constraints $\int xy\, d\mu \geq \frac{1}{2}$ and $\int yz\, d\mu + \int z\, d\mu \geq 1$ to the previous example:

```
from sympy import *
# introduce variables
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
# initiate the Relaxation object
Rlx = SDPRelaxations([x, y, z])
```

```
# set the objective
Rlx.SetObjective(-2 * x + y - z)
# add support constraints
Rlx.AddConstraint(24 - 20 * x + 9 * y - 13 * z + 4 * x**2 -
                  4 * x * y + 4 * x * z + 2 * y**2 - 2 * y * z + 2 * z**2 >= 0)
Rlx.AddConstraint(x + y + z <= 4)
Rlx.AddConstraint(3 * y + z <= 6)
Rlx.AddConstraint(x >= 0)
Rlx.AddConstraint(x <= 2)
Rlx.AddConstraint(y >= 0)
Rlx.AddConstraint(z >= 0)
Rlx.AddConstraint(z <= 3)
# add moment constraints
Rlx.MomentConstraint(Mom(x * y) >= .5)
Rlx.MomentConstraint(Mom(y * z) + Mom(z) >= 1)
# set the relaxation order
Rlx.MomentsOrd(3)
# set the solver
Rlx.SetSDPSolver('dsdp')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
# output
print Rlx.Solution
print "Moment of x*y:", Rlx.Solution.TruncatedMmntSeq[x * y]
print "Moment of y*z + z:", Rlx.Solution.TruncatedMmntSeq[y * z] + Rlx.Solution.
↪TruncatedMmntSeq[z]
```

Solution is:

```
Solution of a Semidefinite Program:
                Solver: DSDP
                Status: Optimal
   Initialization Time: 32.9848718643 seconds
              Run Time: 1.041935 seconds
Primal Objective Value: -4.03644346623
  Dual Objective Value: -4.03644340796
Feasible solution for moments of order 3

Moment of x*y: 0.500000001712
Moment of y*z + z: 2.72623169152
```

Optimization over Varieties

Now we employ the results of *[GIKM]* to solve more complex optimization problems. The main idea is to represent the given function space as a quotient of a suitable polynomial algebra.

Suppose that we want to optimize the function $\sqrt[3]{(xy)^2} - x + y^2$ over the closed disk with radius 3. In order to deal with the term $\sqrt[3]{(xy)^2}$, we introduce an algebraic relation to `SDPRelaxations` object and give a monomial order for Groebner basis computations (default is *lex* for lexicographic order). Clearly $xy - \sqrt[3]{(xy)}^3 = 0$. Therefore by introducing an auxiliary variable or function symbol, say $f(x, y)$ the problem can be stated in the quotient of $\frac{\mathbb{R}[x,y,f]}{\langle xy - f^3 \rangle}$. To check the result of `SDPRelaxations` we employ `scipy.optimize.minimize` with two solvers `COBYLA` and `COBYLA` and a PSO minimizer like pyswarm:

```
from sympy import *
from Irene import *
# introduce variables
x = Symbol('x')
y = Symbol('y')
f = Function('f')(x, y)
# define algebraic relations
rel = [x * y - f**3]
# initiate the Relaxation object
Rlx = SDPRelaxations([x, y, f], rel)
# set the monomial order
Rlx.SetMonoOrd('lex')
# set the objective
Rlx.SetObjective(f**2 - x + y**2)
# add support constraints
Rlx.AddConstraint(9 - x**2 - y**2 >= 0)
# set the solver
Rlx.SetSDPSolver('cvxopt')
# Rlx.MomentsOrd(2)
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
# output
print Rlx.Solution
# using scipy
from scipy.optimize import minimize
fun = lambda x: numpy.power(x[0]**2 * x[1]**2, 1. / 3.) - x[0] + x[1]**2
cons = (
    {'type': 'ineq', 'fun': lambda x: 9 - x[0]**2 - x[1]**2})
sol1 = minimize(fun, (0, 0), method='COBYLA', constraints=cons)
sol2 = minimize(fun, (0, 0), method='SLSQP', constraints=cons)
print "solution according to 'COBYLA'"
print sol1
print "solution according to 'SLSQP'"
print sol2
# particle swarm optimization
from pyswarm import pso
lb = [-9, -9]
ub = [9, 9]
cns = [cons['fun']]
print "PSO:"
print pso(fun, lb, ub, ieqcons=cns)
```

The output will be:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 0.12473487854 seconds
              Run Time: 0.004865 seconds
Primal Objective Value: -2.99999997394
  Dual Objective Value: -2.9999999473
Feasible solution for moments of order 1

solution according to 'COBYLA'
     fun: -0.99788411120450926
   maxcv: 0.0
```

```
 message: 'Optimization terminated successfully.'
    nfev: 25
  status: 1
 success: True
       x: array([  9.99969494e-01,   9.52333693e-05])
 solution according to 'SLSQP'
     fun: -2.9999975825413681
     jac: array([  -0.99999923,  689.00398242,    0.        ])
 message: 'Optimization terminated successfully.'
    nfev: 64
     nit: 13
    njev: 13
  status: 0
 success: True
       x: array([  3.00000000e+00,  -1.25290367e-09])
PSO:
Stopping search: Swarm best position change less than 1e-08
(array([  2.99999996e+00,   4.41523681e-12]), -2.9999999060604554)
```

Optimization over arbitrary functions

Any given algebra can be represented as a quotient of a suitable polynomial algebra (on possibly infinitely many variables). Since optimization problems usually involve finitely many functions and constraints, we can apply the technique introduced in the previous section, as soon as we figure out the quotient representation of the function space.

Let us walk through the procedure by solving some examples.

Example 1. Find the optimum value of the following program:

$$\begin{cases} \min & -(\sin(x)-1)^3-(\sin(x)-\cos(y))^4-(\cos(y)-3)^2 \\ \text{subject to} & \\ & 10-(\sin(x)-1)^2\geq 0, \\ & 10-(\sin(x)-\cos(y))^2\geq 0, \\ & 10-(\cos(y)-3)^2\geq 0. \end{cases}$$

Let us introduce four symbols to represent trigonometric functions:

$f:$	$\sin(x)$	$g:$	$\cos(x)$
$h:$	$\sin(y)$	$k:$	$\cos(y)$

Then the quotient algebra $\frac{\mathbb{R}[f,g,h,k]}{I}$ where $I=\langle f^2+g^2-1, h^2+k^2-1\rangle$ is the right framework to solve the optimization problem. We also compare the outcome of `SDPRelaxations` with `scipy` and `pyswarm`:

```
from sympy import *
from Irene import *
# introduce variables
x = Symbol('x')
f = Function('f')(x)
g = Function('g')(x)
h = Function('h')(x)
k = Function('k')(x)
# define algebraic relations
rels = [f**2 + g**2 - 1, h**2 + k**2 - 1]
# initiate the Relaxation object
```

```
Rlx = SDPRelaxations([f, g, h, k], rels)
# set the monomial order
Rlx.SetMonoOrd('lex')
# set the objective
Rlx.SetObjective(-(f - 1)**3 - (f - k)**4 - (k - 3)**2)
# add support constraints
Rlx.AddConstraint(10 - (f - 1)**2 >= 0)
Rlx.AddConstraint(10 - (f - k)**2 >= 0)
Rlx.AddConstraint(10 - (k - 3)**2 >= 0)
# set the solver
Rlx.SetSDPSolver('csdp')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
# output
print Rlx.Solution
# using scipy
from scipy.optimize import minimize
fun = lambda x: -(sin(x[0]) - 1)**3 - (sin(x[0]) -
                                       cos(x[1]))**4 - (cos(x[1]) - 3)**2
cons = (
    {'type': 'ineq', 'fun': lambda x: 10 - (sin(x[0]) - 1)**2},
    {'type': 'ineq', 'fun': lambda x: 10 - (sin(x[0]) - cos(x[1]))**2},
    {'type': 'ineq', 'fun': lambda x: 10 - (cos(x[1]) - 3)**2})
sol1 = minimize(fun, (0, 0), method='COBYLA', constraints=cons)
sol2 = minimize(fun, (0, 0), method='SLSQP', constraints=cons)
print  "solution according to 'COBYLA':"
print sol1
print "solution according to 'SLSQP':"
print sol2
# particle swarm optimization
from pyswarm import pso
lb = [-10, -10]
ub = [10, 10]
cns = [cons[0]['fun'], cons[1]['fun'], cons[2]['fun']]
print "PSO:"
print pso(fun, lb, ub, ieqcons=cns)
```

Solutions are:

```
Solution of a Semidefinite Program:
                Solver: CSDP
                Status: Optimal
   Initialization Time: 4.23285317421 seconds
              Run Time: 0.016662 seconds
Primal Objective Value: -12.0
  Dual Objective Value: -12.0
Feasible solution for moments of order 2

solution according to 'COBYLA':
     fun: -11.824901993777621
   maxcv: 1.7763568394002505e-15
 message: 'Optimization terminated successfully.'
    nfev: 42
  status: 1
 success: True
       x: array([ 1.57064986,  1.7337948 ])
```

```
solution according to 'SLSQP':
     fun: -11.9999999999720
     jac: array([ -2.94446945e-05,  -1.78813934e-05,   0.00000000e+00])
 message: 'Optimization terminated successfully.'
    nfev: 23
     nit: 5
    njev: 5
  status: 0
 success: True
       x: array([ -1.57079782e+00,  -6.42618794e-07])
PSO:
Stopping search: Swarm best objective change less than 1e-08
(array([-1.57078003,  6.28318074]), -11.999999997051434)
```

CHAPTER

FOUR

APPROXIMATING OPTIMUM VALUE

In various cases, separating functions and symbols are either very difficult or impossible. For example $x, \sin x$ and e^x are not algebraically independent, but their dependency can not be easily expressed in finitely many relations. One possible approach to these problems is replacing transcendental terms with a reasonably good approximation. This certainly will introduce more numerical error, but at least gives a reliable estimate for the optimum value.

Using `pyProximation.OrthSystem`

A simple and common method to approximate transcendental functions is using truncated Taylor expansions. In spite of its simplicity, there are various pitfalls which needs to be avoided. The most common is that the radius of convergence of the Taylor expansion may be smaller than the feasibility region of the optimization problem.

Example 1:

Find the minimum of $x + e^{x \sin x}$ *where* $-\pi \leq x \leq \pi$.

The objective function includes terms of x and transcendental functions. So, it is difficult to find a suitable algebraic representation to transform this optimization problem. Let us try to use Taylor expansion of $e^{x \sin x}$ to find an approximation for the optimum and compare the result with `scipy.optimize` and `pyswarm.pso`:

```
from sympy import *
from Irene import *
# introduce symbols and functions
x = Symbol('x')
e = Function('e')(x)
# transcendental term of objective
f = exp(x * sin(x))
# Taylor expansion
f_app = f.series(x, 0, 12).removeO()
# initiate the Relaxation object
Rlx = SDPRelaxations([x])
# set the objective
Rlx.SetObjective(x + f_app)
# add support constraints
Rlx.AddConstraint(pi**2 - x**2 >= 0)
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
# using scipy
from scipy.optimize import minimize
```

```
fun = lambda x: x[0] + exp(x[0] * sin(x[0]))
cons = (
    {'type': 'ineq', 'fun': lambda x: pi**2 - x[0]**2},
)
sol1 = minimize(fun, (0, 0), method='COBYLA', constraints=cons)
sol2 = minimize(fun, (0, 0), method='SLSQP', constraints=cons)
print "solution according to 'COBYLA':"
print sol1
print "solution according to 'SLSQP':"
print sol2
# particle swarm optimization
from pyswarm import pso
lb = [-3.3, -3.3]
ub = [3.3, 3.3]
cns = [cons[0]['fun']]
print "PSO:"
print pso(fun, lb, ub, ieqcons=cns)
```

The output will look like:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 0.270121097565 seconds
              Run Time: 0.012974 seconds
Primal Objective Value: -416.628918881
  Dual Objective Value: -416.628917197
Feasible solution for moments of order 5

solution according to 'COBYLA':
     fun: 0.76611902154788758
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 34
  status: 1
 success: True
       x: array([ -4.42161128e-01,  -9.76206736e-05])
solution according to 'SLSQP':
     fun: 0.766119450232887
     jac: array([ 0.00154828,  0.        ,  0.        ])
 message: 'Optimization terminated successfully.'
    nfev: 17
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([-0.44164406,  0.        ])
PSO:
Stopping search: Swarm best objective change less than 1e-08
(array([-3.14159265,  1.94020281]), -2.1415926274654815)
```

Now instead of Taylor expansion, we use Legendre polynomials to estimate $e^{x \sin x}$:

```
from sympy import *
from Irene import *
from pyProximation import OrthSystem
# introduce symbols and functions
x = Symbol('x')
```

```
e = Function('e')(x)
# transcendental term of objective
f = exp(x * sin(x))
# Legendre polynomials via pyProximation
D = [(-pi, pi)]
S = OrthSystem([x], D)
# set B = {1, x, x^2, ..., x^12}
B = S.PolyBasis(12)
# link B to S
S.Basis(B)
# generate the orthonormal basis
S.FormBasis()
# extract the coefficients of approximation
Coeffs = S.Series(f)
# form the approximation
f_app = sum([S.OrthBase[i] * Coeffs[i] for i in range(len(S.OrthBase))])
# initiate the Relaxation object
Rlx = SDPRelaxations([x])
# set the objective
Rlx.SetObjective(x + f_app)
# add support constraints
Rlx.AddConstraint(pi**2 - x**2 >= 0)
# set the solver
Rlx.SetSDPSolver('dsdp')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
```

The output will be:

```
Solution of a Semidefinite Program:
                Solver: DSDP
                Status: Optimal
   Initialization Time: 0.722383022308 seconds
              Run Time: 0.077674 seconds
Primal Objective Value: -2.26145824829
  Dual Objective Value: -2.26145802066
Feasible solution for moments of order 6
```

By a small modification of the above code, we can employ Chebyshev polynomials for approximation:

```
from sympy import *
from Irene import *
from pyProximation import Measure, OrthSystem
# introduce symbols and functions
x = Symbol('x')
e = Function('e')(x)
# transcendental term of objective
f = exp(x * sin(x))
# Chebyshev polynomials via pyProximation
D = [(-pi, pi)]
# the Chebyshev weight
w = lambda x: 1. / sqrt(pi**2 - x**2)
M = Measure(D, w)
S = OrthSystem([x], D)
# link the measure to S
```

```
S.SetMeasure(M)
# set B = {1, x, x^2, ..., x^12}
B = S.PolyBasis(12)
# link B to S
S.Basis(B)
# generate the orthonormal basis
S.FormBasis()
# extract the coefficients of approximation
Coeffs = S.Series(f)
# form the approximation
f_app = sum([S.OrthBase[i] * Coeffs[i] for i in range(len(S.OrthBase))])
# initiate the Relaxation object
Rlx = SDPRelaxations([x])
# set the objective
Rlx.SetObjective(x + f_app)
# add support constraints
Rlx.AddConstraint(pi**2 - x**2 >= 0)
# set the solver
Rlx.SetSDPSolver('dsdp')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
```

which returns:

```
Solution of a Semidefinite Program:
                Solver: DSDP
                Status: Optimal
   Initialization Time: 0.805300951004 seconds
              Run Time: 0.066767 seconds
Primal Objective Value: -2.17420785198
  Dual Objective Value: -2.17420816422
Feasible solution for moments of order 6
```

This gives a better approximation for the optimum value. The optimum values found via Legendre and Chebyshev polynomials are certainly better than Taylor expansion and the results of `scipy.optimize`.

Example 2:

Find the minimum of $x \sinh y + e^{y \sin x}$ *where* $-\pi \leq x, y \leq \pi$.

Again, we use Legendre approximations for $\sinh y$ and $e^{y \sin x}$:

```
from sympy import *
from Irene import *
from pyProximation import OrthSystem
# introduce symbols and functions
x = Symbol('x')
y = Symbol('y')
sh = Function('sh')(y)
ch = Function('ch')(y)
# transcendental term of objective
f = exp(y * sin(x))
g = sinh(y)
# Legendre polynomials via pyProximation
```

```
D_f = [(-pi, pi), (-pi, pi)]
D_g = [(-pi, pi)]
Orth_f = OrthSystem([x, y], D_f)
Orth_g = OrthSystem([y], D_g)
# set bases
B_f = Orth_f.PolyBasis(10)
B_g = Orth_g.PolyBasis(10)
# link B to S
Orth_f.Basis(B_f)
Orth_g.Basis(B_g)
# generate the orthonormal bases
Orth_f.FormBasis()
Orth_g.FormBasis()
# extract the coefficients of approximations
Coeffs_f = Orth_f.Series(f)
Coeffs_g = Orth_g.Series(g)
# form the approximations
f_app = sum([Orth_f.OrthBase[i] * Coeffs_f[i]
             for i in range(len(Orth_f.OrthBase))])
g_app = sum([Orth_g.OrthBase[i] * Coeffs_g[i]
             for i in range(len(Orth_g.OrthBase))])
# initiate the Relaxation object
Rlx = SDPRelaxations([x, y])
# set the objective
Rlx.SetObjective(x * g_app + f_app)
# add support constraints
Rlx.AddConstraint(pi**2 - x**2 >= 0)
Rlx.AddConstraint(pi**2 - y**2 >= 0)
# set the sdp solver
Rlx.SetSDPSolver('cvxopt')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
# using scipy
from scipy.optimize import minimize
fun = lambda x: x[0] * sinh(x[1]) + exp(x[1] * sin(x[0]))
cons = (
    {'type': 'ineq', 'fun': lambda x: pi**2 - x[0]**2},
    {'type': 'ineq', 'fun': lambda x: pi**2 - x[1]**2}
)
sol1 = minimize(fun, (0, 0), method='COBYLA', constraints=cons)
sol2 = minimize(fun, (0, 0), method='SLSQP', constraints=cons)
print "solution according to 'COBYLA':"
print sol1
print "solution according to 'SLSQP':"
print sol2
# particle swarm optimization
from pyswarm import pso
lb = [-3.3, -3.3]
ub = [3.3, 3.3]
cns = [cons[0]['fun'], cons[1]['fun']]
print "PSO:"
print pso(fun, lb, ub, ieqcons=cns)
```

The result will be:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 18.4279370308 seconds
              Run Time: 0.123869 seconds
Primal Objective Value: -35.3574475835
  Dual Objective Value: -35.3574473266
Feasible solution for moments of order 5

solution according to 'COBYLA':
     fun: 1.0
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 13
  status: 1
 success: True
       x: array([ 0.,  0.])
solution according to 'SLSQP':
     fun: 1
     jac: array([ 0.,  0.,  0.])
 message: 'Optimization terminated successfully.'
    nfev: 4
     nit: 1
    njev: 1
  status: 0
 success: True
       x: array([ 0.,  0.])
PSO:
Stopping search: Swarm best position change less than 1e-08
(array([ 3.14159265, -3.14159265]), -35.281434655101151)
```

which shows a significant improvement compare to results of `scipi.minimize`.

CHAPTER

FIVE

BENCHMARK OPTIMIZATION PROBLEMS

There are benchmark problems to evaluated how good an optimization method works. We apply the generalized relaxation method to some of these benchmarks that are mainly taken from *[MJXY]*.

Rosenbrock Function

The original Rosenbrock function is $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ which is a sums of squares and attains its minimum at $(1, 1)$. The global minimum is inside a long, narrow, parabolic shaped flat valley. To find the valley is trivial. To converge to the global minimum, however, is difficult. The same holds for a generalized form of Rosenbrock function which is defined as:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2.$$

Since f is a sum of squares, and $f(1, \ldots, 1) = 0$, the global minimum is equal to 0. The following code compares various optimization methods including the relaxation method, to find a minimum for f where $9 - x_i^2 \geq 0$ for $i = 1, \ldots, 9$:

```
from sympy import *
from Irene import SDPRelaxations
NumVars = 9
# define the symbolic variables and functions
x = [Symbol('x%d' % i) for i in range(NumVars)]

print "Relaxation method:"
# initiate the SDPRelaxations object
Rlx = SDPRelaxations(x)
# Rosenbrock function
rosen = sum([100 * (x[i + 1] - x[i]**2)**2 + (1 - x[i])
             ** 2 for i in range(NumVars - 1)])
# set the objective
Rlx.SetObjective(rosen)
# add constraints
for i in range(NumVars):
    Rlx.AddConstraint(9 - x[i]**2 >= 0)
# set the sdp solver
Rlx.SetSDPSolver('cvxopt')
# initiate the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
```

```
# solve with scipy
from scipy.optimize import minimize
fun = lambda x: sum([100 * (x[i + 1] - x[i]**2)**2 +
                     (1 - x[i])**2 for i in range(NumVars - 1)])
cons = [
    {'type': 'ineq', 'fun': lambda x: 9 - x[i]**2} for i in range(NumVars)]
x0 = tuple([0 for _ in range(NumVars)])
print  "solution according to 'COBYLA':"
sol1 = minimize(fun, x0, method='COBYLA', constraints=cons)
print "solution according to 'SLSQP':"
sol2 = minimize(fun, x0, method='SLSQP', constraints=cons)
print sol1
print sol2
# solve with pso
print "PSO:"
from pyswarm import pso
lb = [-3 for i in range(NumVars)]
ub = [3 for i in range(NumVars)]
cns = [cn['fun'] for cn in cons]
print pso(fun, lb, ub, ieqcons=cns)
```

The result is:

```
Relaxation method:
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 750.234924078 seconds
              Run Time: 8.43369 seconds
Primal Objective Value: 1.67774267808e-08
  Dual Objective Value: 1.10015692778e-08
Feasible solution for moments of order 2

solution according to 'COBYLA':
     fun: 4.4963584556077389
   maxcv: 0.0
 message: 'Maximum number of function evaluations has been exceeded.'
    nfev: 1000
  status: 2
 success: False
       x: array([  8.64355944e-01,   7.47420978e-01,   5.59389194e-01,
         3.16212252e-01,   1.05034350e-01,   2.05923923e-02,
         9.44389237e-03,   1.12341021e-02,  -7.74530516e-05])
     fun: 1.3578865444308464e-07
     jac: array([ 0.00188377,  0.00581741, -0.00182463,  0.00776938, -0.00343305,
       -0.00186283,  0.0020364 ,  0.00881489, -0.0047164 ,  0.        ])
solution according to 'SLSQP':
 message: 'Optimization terminated successfully.'
    nfev: 625
     nit: 54
    njev: 54
  status: 0
 success: True
       x: array([ 1.00000841,  1.00001216,  1.00000753,  1.00001129,  1.00000134,
        1.00000067,  1.00000502,  1.00000682,  0.99999006])
PSO:
Stopping search: maximum iterations reached --> 100
(array([-0.30495496,  0.10698904, -0.129344  ,  0.07972014,  0.027356  ,
```

```
        0.02170117, -0.0036854 ,  0.10778454,  0.04141022]), 12.067752160169965)
```

The relaxation method returns values very close to the actual minimum but two out of other three methods fail to estimate the minimum correctly.

Giunta Function

Giunta is an example of continuous, differentiable, separable, scalable, multimodal function defined by:

$$\begin{aligned} f(x_1, x_2) &= \frac{3}{5} + \sum_{i=1}^{2}[\sin(\frac{16}{15}x_i - 1) \\ &+ \sin^2(\frac{16}{15}x_i - 1) \\ &+ \frac{1}{50}\sin(4(\frac{16}{15}x_i - 1))]. \end{aligned}$$

The following code optimizes f when $1 - x^2 \geq 0$ and $1 - y^2 \geq 0$:

```
from sympy import *
from Irene import *
x = Symbol('x')
y = Symbol('y')
s1 = Symbol('s1')
c1 = Symbol('c1')
s2 = Symbol('s2')
c2 = Symbol('c2')
f = .6 + (sin(x - 1) + (sin(x - 1))**2 + .02 * sin(4 * (x - 1))) + \
    (sin(y - 1) + (sin(y - 1))**2 + .02 * sin(4 * (y - 1)))
f = expand(f, trig=True)
f = N(f.subs({sin(x): s1, cos(x): c1, sin(y): s2, cos(y): c2}))
rels = [s1**2 + c1**2 - 1, s2**2 + c2**2 - 1]
Rlx = SDPRelaxations([s1, c1, s2, c2], rels)
Rlx.SetObjective(f)
Rlx.AddConstraint(1 - s1**2 >= 0)
Rlx.AddConstraint(1 - s2**2 >= 0)
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
# solve with scipy
from scipy.optimize import minimize
fun = lambda x: .6 + (sin((16. / 15.) * x[0] - 1) + (sin((16. / 15.) * x[0] - 1))**2␣
↪+ .02 * sin(4 * ((16. / 15.) * x[0] - 1))) + (
    sin((16. / 15.) * x[1] - 1) + (sin((16. / 15.) * x[1] - 1))**2 + .02 * sin(4 *␣
↪((16. / 15.) * x[1] - 1)))
cons = [
    {'type': 'ineq', 'fun': lambda x: 1 - x[i]**2} for i in range(2)]
x0 = tuple([0 for _ in range(2)])
sol1 = minimize(fun, x0, method='COBYLA', constraints=cons)
sol2 = minimize(fun, x0, method='SLSQP', constraints=cons)
print  "solution according to 'COBYLA':"
print sol1
print "solution according to 'SLSQP':"
print sol2
# solve with pso
print "PSO:"
from pyswarm import pso
lb = [-1 for i in range(2)]
ub = [1 for i in range(2)]
```

```
cns = [cn['fun'] for cn in cons]
print pso(fun, lb, ub, ieqcons=cns)
```

and the result is:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 3.03993797302 seconds
              Run Time: 0.015762 seconds
Primal Objective Value: 0.0644704534329
  Dual Objective Value: 0.0644704595475
Feasible solution for moments of order 2

solution according to 'COBYLA':
     fun: 0.064470433766030344
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 45
  status: 1
 success: True
       x: array([ 0.49835509,  0.49847982])
solution according to 'SLSQP':
     fun: 0.0644705317528075
     jac: array([ 0.00045323,  0.00045323,  0.        ])
 message: 'Optimization terminated successfully.'
    nfev: 17
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([ 0.4987201,  0.4987201])
PSO:
Stopping search: Swarm best objective change less than 1e-08
(array([ 0.49858097,  0.49835221]), 0.064470444814555605)
```

Parsopoulos Function

Parsopoulos is defined as $f(x,y) = \cos^2(x) + \sin^2(y)$. The following code computes its minimum where $-5 \leq x, y \leq 5$:

```
from sympy import *
from Irene import *
x = Symbol('x')
y = Symbol('y')
s1 = Symbol('s1')
c1 = Symbol('c1')
s2 = Symbol('s2')
c2 = Symbol('c2')
f = c1**2 + s2**2
rels = [s1**2 + c1**2 - 1, s2**2 + c2**2 - 1]
Rlx = SDPRelaxations([s1, c1, s2, c2], rels)
Rlx.SetObjective(f)
Rlx.AddConstraint(1 - s1**2 >= 0)
Rlx.AddConstraint(1 - s2**2 >= 0)
```

```
Rlx.MomentsOrd(2)
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
# solve with scipy
from scipy.optimize import minimize
fun = lambda x: cos(x[0])**2 + sin(x[1])**2
cons = [
    {'type': 'ineq', 'fun': lambda x: 25 - x[i]**2} for i in range(2)]
x0 = tuple([0 for _ in range(2)])
sol1 = minimize(fun, x0, method='COBYLA', constraints=cons)
sol2 = minimize(fun, x0, method='SLSQP', constraints=cons)
print "solution according to 'COBYLA':"
print sol1
print "solution according to 'SLSQP':"
print sol2
# solve with pso
print "PSO:"
from pyswarm import pso
lb = [-5 for i in range(2)]
ub = [5 for i in range(2)]
cns = [cn['fun'] for cn in cons]
print pso(fun, lb, ub, ieqcons=cns)
```

which returns:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 3.01474308968 seconds
              Run Time: 0.013299 seconds
Primal Objective Value: -3.7471929546e-10
  Dual Objective Value: 5.43046022792e-12
Feasible solution for moments of order 2

solution according to 'COBYLA':
     fun: 1.8371674257900859e-08
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 35
  status: 1
 success: True
       x: array([  1.57072551e+00,   1.15569800e-04])
solution according to 'SLSQP':
     fun: 1
     jac: array([ -1.49011612e-08,   1.49011612e-08,   0.00000000e+00])
 message: 'Optimization terminated successfully.'
    nfev: 4
     nit: 1
    njev: 1
  status: 0
 success: True
       x: array([ 0.,  0.])
PSO:
Stopping search: Swarm best objective change less than 1e-08
(array([ 4.71233715,  3.14155673]), 3.9770280273184657e-09)
```

Shubert Function

Subert function is defined by:

$$f(x_1, \ldots, x_n) = \prod_{i=1}^{n} \left(\sum_{j=1}^{5} \cos((j+1)x_i + i) \right).$$

It is a continuous, differentiable, separable, non-scalable, multimodal function. The following code compares the result of five optimizers when $-10 \leq x_i \leq 10$ and $n = 2$:

```
from sympy import *
from Irene import *
x = Symbol('x')
y = Symbol('y')
s1 = Symbol('s1')
c1 = Symbol('c1')
s2 = Symbol('s2')
c2 = Symbol('c2')
f = sum([cos((j + 1) * x + j) for j in range(1, 6)]) * \
    sum([cos((j + 1) * y + j) for j in range(1, 6)])
obj = N(expand(f, trig=True).subs(
    {sin(x): s1, cos(x): c1, sin(y): s2, cos(y): c2}))
rels = [s1**2 + c1**2 - 1, s2**2 + c2**2 - 1]
Rlx = SDPRelaxations([s1, c1, s2, c2], rels)
Rlx.SetObjective(obj)
Rlx.AddConstraint(1 - s1**2 >= 0)
Rlx.AddConstraint(1 - s2**2 >= 0)
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
g = lambda x: sum([cos((j + 1) * x[0] + j) for j in range(1, 6)]) * \
    sum([cos((j + 1) * x[1] + j) for j in range(1, 6)])
x0 = (-5, 5)
from scipy.optimize import minimize
cons = (
    {'type': 'ineq', 'fun': lambda x: 100 - x[0]**2},
    {'type': 'ineq', 'fun': lambda x: 100 - x[1]**2})
sol1 = minimize(g, x0, method='COBYLA', constraints=cons)
sol2 = minimize(g, x0, method='SLSQP', constraints=cons)
print "solution according to 'COBYLA':"
print sol1
print "solution according to 'SLSQP':"
print sol2

from sage.all import *
m1 = minimize_constrained(g, cons=[cn['fun'] for cn in cons], x0=x0)
m2 = minimize_constrained(g, cons=[cn['fun']
                                   for cn in cons], x0=x0, algorithm='l-bfgs-b')
print "Sage:"
print "minimize_constrained (default):", m1, g(m1)
print "minimize_constrained (l-bfgs-b):", m2, g(m2)
print "PSO:"
from pyswarm import pso
lb = [-10, -10]
ub = [10, 10]
cns = [cn['fun'] for cn in cons]
```

```
print pso(g, lb, ub, ieqcons=cns)
```

The result is:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 1129.02412415 seconds
              Run Time: 5.258507 seconds
Primal Objective Value: -18.0955649723
  Dual Objective Value: -18.0955648855
Feasible solution for moments of order 6
Scipy 'COBYLA':
     fun: -3.3261182321238367
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 39
  status: 1
 success: True
       x: array([-3.96201407,  4.81176624])
Scipy 'SLSQP':
     fun: -0.856702387212005
     jac: array([-0.00159422,  0.00080796,  0.        ])
 message: 'Optimization terminated successfully.'
    nfev: 35
     nit: 7
    njev: 7
  status: 0
 success: True
       x: array([-4.92714381,  4.81186391])
Sage:
minimize_constrained (default): (-3.962032420336303, 4.811734682897321) -3.32611819422
minimize_constrained (l-bfgs-b): (-3.962032420336303, 4.811734682897321) -3.
↪32611819422
PSO:
Stopping search: Swarm best objective change less than 1e-08
(array([-0.77822054,  4.8118272 ]), -18.095565036766224)
```

We note that four out of five optimizers stuck at a local minimum and return incorrect values.

CHAPTER

SIX

CODE DOCUMENTATION

class base.**base**

All the modules in *Irene* extend this class which perform some common tasks such as checking existence of certain softwares.

AvailableSDPSolvers()

find the existing sdp solvers.

which(*program*)

Check the availability of the *program* system-wide. Returns the path of the program if exists and returns 'None' otherwise.

class relaxations.**Mom**(*expr*)

This is a simple interface to define moment constraints to be used via *SDPRelaxations.MomentConstraint*. It takes a sympy expression as input and initiates an object which can be used to force particular constraints on the moment sequence.

Example: Force the moment of $x^2 f(x) + f(x)^2$ to be at least *.5*:

```
Mom(x**2 * f + f**2) >= .5
# OR
Mom(x**2 * f) + Mom(f**2) >= .5
```

class relaxations.**SDPRelaxations**(*gens*, *relations=[]*)

This class defines a function space by taking a family of sympy symbolic functions and relations among them. Simply, it initiates a commutative free real algebra on the symbolic functions and defines the function space as the quotient of the free algebra by the ideal generated by the given relations. It takes two arguments:

- *gens* which is a list of sympy symbols and function symbols,
- *relations* which is a set of sympy expressions in terms of *gens* that defines an ideal.

AddConstraint(*cnst*)

Takes an (in)equality as an algebraic combination of the generating functions that defines the feasibility region. It reduces the defining (in)equalities according to the given relations.

Calpha(*expn*, *Mmnt*)

Given an exponent *expn*, this method finds the corresponding C_{expn} matrix.

ExponentsVec(*deg*)

Returns all the exponents that appear in the reduced basis of all monomials of the auxiliary symbols of degree at most *deg*.

InitSDP()

Initializes the semidefinite program (SDP) whose solution is a lower bound for the minimum of the program.

LocalizedMoment(*p*)
Computes the reduced symbolic moment generating matrix localized at *p*.

Minimize()
Finds the minimum of the truncated moment problem which provides a lower bound for the actual minimum.

MomentConstraint(*cnst*)
Takes constraints on the moments. The input must be an instance of *Mom* class.

MomentMat()
Returns the numerical moment matrix resulted from solving the SDP.

MomentsOrd(*ord*)
Sets the order of moments to be considered.

PolyCoefFullVec()
return the vector of coefficient of the reduced objective function as an element of the vector space of elements of degree up to the order of moments.

ReduceExp(*expr*)
Takes an expression *expr*, either in terms of internal free symbolic variables or generating functions and returns the reduced expression in terms of internal symbolic variables, if a relation among generators is present, otherwise it just substitutes generating functions with their corresponding internal symbols.

ReducedMonomialBase(*deg*)
Returns a reduce monomial basis up to degree *d*.

RelaxationDeg()
Finds the minimum required order of moments according to user's request, objective function and constraints.

SetMonoOrd(*ord*)
Changes the default monomial order to *ord* which mustbe among *lex*, *grlex*, *grevlex*, *ilex*, *igrlex*, *igrevlex*.

SetObjective(*obj*)
Takes the objective function *obj* as an algebraic combination of the generating symbolic functions, replace the symbolic functions with corresponding auxiliary symbols and reduce them according to the given relations.

SetSDPSolver(*solver*)

Sets the default SDP solver. The followings are currently supported:

- CVXOPT
- DSDP
- SDPA
- CSDP

The selected solver must be installed otherwise it cannot be called. The default solver is *CVXOPT* which has an interface for Python. *DSDP* is called through the CVXOPT's interface. *SDPA* and *CSDP* are called independently.

class relaxations.**SDRelaxSol**
Instances of this class carry information on the solution of the semidefinite relaxation associated to a optimization problem. It include various pieces of information:

- `SDRelaxSol.TruncatedMmntSeq` a dictionary of resulted moments
- `SDRelaxSol.MomentMatrix` the resulted moment matrix

- `SDRelaxSol.Primal` the value of the SDP in primal form
- `SDRelaxSol.Dual` the value of the SDP in dual form
- `SDRelaxSol.RunTime` the run time of the sdp solver
- `SDRelaxSol.InitTime` the total time consumed for initialization of the sdp
- `SDRelaxSol.Solver` the name of sdp solver
- `SDRelaxSol.Status` final status of the sdp solver
- `SDRelaxSol.RelaxationOrd` order of relaxation
- `SDRelaxSol.Message` the message that maybe returned by the sdp solver

class `sdp.`**sdp**(*solver='cvxopt'*)

This is the class which intends to solve semidefinite programs in primal format:

$$\begin{cases} \min & \sum_{i=1}^{m} b_i x_i \\ \text{subject to} & \\ & \sum_{i=1}^{m} A_{ij} x_i - C_j \succeq 0 \quad j = 1, \ldots, k. \end{cases}$$

For the argument *solver* following sdp solvers are supported (if they are installed):

- *CVXOPT*,
- *CSDP*,
- *SDPA*,
- *DSDP*.

AddConstantBlock(*C*)

C must be a list of `numpy` matrices that represent C_j for each *j*. This method sets the value for $C = [C_1, \ldots, C_k]$.

AddConstraintBlock(*A*)

This takes a list of square matrices which corresponds to coefficient of x_i. Simply, $A_i = [A_{i1}, \ldots, A_{ik}]$. Note that the i^{th} call of `AddConstraintBlock` fills the blocks associated with i^{th} variable x_i.

CvxOpt()

This calls *CVXOPT* and *DSDP* to solve the initiated semidefinite program.

Option(*param*, *val*)

Sets the *param* option of the solver to *val* if the solver accepts such an option.

SetObjective(*b*)

Takes the coefficients of the objective function.

VEC(*M*)

Converts the matrix M into a column vector acceptable by *CVXOPT*.

csdp()

Calls *SDPA* to solve the initiated semidefinite program.

parse_solution_matrix(*iterator*)

Parses and returns the matrices and vectors found by *SDPA* solver. This was taken from *ncpol2sdpa* and customized for *Irene*.

read_csdp_out(*filename*, *txt*)

Takes a file name and a string that are the outputs of *CSDP* as a file and command line outputs of the solver and extracts the required information.

read_sdpa_out(*filename*)
Extracts information from *SDPA*'s output file *filename*. This was taken from *ncpol2sdpa* and customized for *Irene*.

sdpa()
Calls *SDPA* to solve the initiated semidefinite program.

solve()
Solves the initiated semidefinite program according to the requested solver.

write_sdpa_dat(*filename*)
Writes the semidefinite program in the file *filename* with dense SDPA format.

write_sdpa_dat_sparse(*filename*)
Writes the semidefinite program in the file *filename* with sparse SDPA format.

CHAPTER

SEVEN

REVISION HISTORY

Version 1.0.0 (Dec 0x, 2016)

- Initial release

CHAPTER

EIGHT

TO DO

Based on the current implementation, the followings seems to be implemented/modified:

- Reduce dependency on SymPy.
- Parallelize `InitSDP()` which is the slowest step.
- Write a `__str__` method for `SDPRelaxations` printing.
- Keep track of original expressions before reduction.
- Write a LaTeX method.
- Include sdp solvers upon installation.
- Error handling for CSDP failure.
- Extract solutions (at least for polynomials).

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

[GIKM] M. Ghasemi, M. Infusino, S. Kuhlmann and M. Marshall, *Truncated Moment Problem for unital commutative real algebras*, to appear.

[JBL] J-B. Lasserre, *Global optimization with polynomials and the problem of moments*, SIAM J. Optim. 11(3) 796-817 (2000).

[MJXY] 13. Jamil, Xin-She Yang, *A literature survey of benchmark functions for global optimization problems*, IJMMNO, Vol. 4(2), 2013.

b

r

s