Irene Documentation

Release 1.1.0

Bahareh Esfahbod & Mehdi Ghasemi

Dec 25, 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.

1.1 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.
  - SciPy.
- **for symbolic computations:**
  - SymPy.
- **for parallel computations (optional):**
  - Joblib.
- **for semidefinite optimization, at least one of the following is required:**
  - cvxopt,
  - dsdp,
  - sdpa,
  - csdp.

1.2 Download

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

1.3 Installation

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

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

1.3.1 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
```

Documentation is also available at http://irene.readthedocs.io.

1.4 License

Irene is distributed under MIT license:

1.4.1 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.

2.1 Primal and Dual formulations

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

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

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

$$\left\{\begin{array}{lll} \max & \sum_{j=1}^k tr(C_j \times Z_j) & \\ \text{subject to} & & \\ & \sum_{j=1}^k tr(A_{ij} \times Z_j) = b_i & i = 1, \ldots, m, \\ & Z_j \succeq 0 & j = 1, \ldots, k. \end{array}\right.$$

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

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

and

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

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 $i = 1, dots, m$.

2.2 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:

2.2.1 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.

2.2.2 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.

2.2.3 SDPA

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

2.2.4 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.
```

2.2.5 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.

2.2.6 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.

2.2.7 Set the constant block *C*:

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

2.2.8 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.

2.2.9 Example:

Consider the following SDP:

$$\left\{\begin{array}{ll} \min & x_1 - x_2 + x_3 \\ \text{subject to} & \\ & \left(\begin{array}{cc} 7 & 11 \\ 11 & -3 \end{array}\right) x_1 + \left(\begin{array}{cc} -7 & 18 \\ 18 & -8 \end{array}\right) x_2 + \left(\begin{array}{cc} 2 & 8 \\ 8 & -1 \end{array}\right) x_3 \succeq \left(\begin{array}{cc} -33 & 9 \\ 9 & -26 \end{array}\right) \\ & \left(\begin{array}{ccc} 21 & 11 & 0 \\ 11 & -10 & -8 \\ 0 & -8 & -5 \end{array}\right) x_1 + \left(\begin{array}{ccc} 0 & -10 & -16 \\ -10 & 10 & 10 \\ -16 & 10 & -3 \end{array}\right) x_2 + \left(\begin{array}{ccc} 5 & -2 & 17 \\ -2 & 6 & -8 \\ 17 & -8 & -6 \end{array}\right) x_3 \succeq \left(\begin{array}{ccc} -14 & -9 & -40 \\ -9 & -91 & -10 \\ -40 & -10 & -15 \end{array}\right) \end{array}\right.$$

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

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

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.

Note: 1. One behavior that distinguishes this method from others is that using SDP relaxations always provides a lower bound for theminimum value of the objective function over the feasibility set. While other methods usually involve evaluation of the objectiveand hence the result is always an upper bound for the minimum.

2. The SDP relaxation method relies on symbolic computations which could be quite costly and slow. Therefore, dealing with rather large problems -although *Irene* takes advantage from multiple cores- can be rather slow.

3.1 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: 8.04711222649 seconds
              Run Time: 1.056733 seconds
Primal Objective Value: -4.06848294478
  Dual Objective Value: -4.06848289445
Feasible solution for moments of order 3
```

3.1.1 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 *
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)
# 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: 7.91646790504 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
```

3.1.2 Equality Constraints

Although it is possible to add equality constraints via `AddConstraint` and `MomentConstraint`, but *SDPRelaxation* converts them to two inequalities and considers a certain margin of error. For $A = B$, it considers $A \geq B - \varepsilon$ and $A \leq B + \varepsilon$. In this case the value of ε can be modified by setting *SDPRelaxation.ErrorTolerance* which its default value is 10^{-6}.

3.2 Optimization of Rational Functions

Given two polynomials $p(X), q(X), g_1(X), \dots, g_m(X)$, the minimum of $\frac{p(X)}{q(X)}$ over $K = \{x : g_i(x) \geq 0,\ i = 1, \dots, m\}$ is equal to

$$\begin{cases} \min & \int p(X)\, d\mu \\ \text{subject to} & \\ & \int q(X)\, d\mu = 1, \\ & \mu \in \mathcal{M}^+(K). \end{cases}$$

Note that in this case μ is not taken to be a probability measure, but instead $\int q(X)\, d\mu = 1$. We can use `SDPRelaxations.Probability = False` to relax the probability condition on μ and use moment constraints to enforce $\int q(X)\, d\mu = 1$. The following example explains this.

Example: Find the minimum of $\frac{x^2-2x}{x^2+2x+1}$:

```
from sympy import *
from Irene import *
# define the symbolic variable
x = Symbol('x')
# initiate the SDPRelaxations object
Rlx = SDPRelaxations([x])
# settings
Rlx.Probability = False
# set the objective
Rlx.SetObjective(x**2 - 2*x)
# moment constraint
Rlx.MomentConstraint(Mom(x**2+2*x+1) == 1)
# set the sdp solver
Rlx.SetSDPSolver('cvxopt')
# initiate the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
```

The result is:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 0.167912006378 seconds
              Run Time: 0.008987 seconds
Primal Objective Value: -0.333333666913
  Dual Objective Value: -0.333333667469
Feasible solution for moments of order 1
```

Note: Beside `SDPRelaxations.Probability` there is another attribute `SDPRelaxations.PSDMoment` which by default is set to `True` and makes sure that the sdp solver assumes positivity for the moment matrix.

3.3 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` as well as two solvers, *Augmented Lagrangian Particle Swarm Optimizer* and *Non Sorting Genetic Algorithm II* from pyOpt:

```
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 numpy import power
from scipy.optimize import minimize
fun = lambda x: 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
```

```
# pyOpt
from pyOpt import *

def objfunc(x):
        from numpy import power
        f = power(x[0]**2 * x[1]**2, 1. / 3.) - x[0] + x[1]**2
        g = [x[0]**2 + x[1]**2 - 9]
        fail = 0
        return f, g, fail

opt_prob = Optimization('A third root function', objfunc)
opt_prob.addVar('x1', 'c', lower=-3, upper=3, value=0.0)
opt_prob.addVar('x2', 'c', lower=-3, upper=3, value=0.0)
opt_prob.addObj('f')
opt_prob.addCon('g1', 'i')
# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

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])

ALPSO Solution to A third root function
================================================================================
```

```
        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                        0.1174
    Total Function Evaluations:          1720
    Lambda: [ 0.00023458]
    Seed: 1482111093.38230896

    Objectives:
        Name        Value        Optimum
             f       -2.99915             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type        Value       Lower Bound  Upper Bound
             x1      c       3.000000      -3.00e+00     3.00e+00
             x2      c       0.000008      -3.00e+00     3.00e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1          i       -1.00e+21 <= 0.000000 <= 0.00e+00

--------------------------------------------------------------------------------


NSGA-II Solution to A third root function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                        0.3833
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f       -2.99898             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type        Value       Lower Bound  Upper Bound
             x1      c       3.000000      -3.00e+00     3.00e+00
             x2      c      -0.000011      -3.00e+00     3.00e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1          i       -1.00e+21 <= -0.000000 <= 0.00e+00

--------------------------------------------------------------------------------
```

3.4 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
# pyOpt
from pyOpt import *


def objfunc(x):
    from numpy import sin, cos
    f = -(sin(x[0]) - 1)**3 - (sin(x[0]) - cos(x[1]))**4 - (cos(x[1]) - 3)**2
    g = [
        (sin(x[0]) - 1)**2 - 10,
        (sin(x[0]) - cos(x[1]))**2 - 10,
        (cos(x[1]) - 3)**2 - 10
    ]
    fail = 0
    return f, g, fail

opt_prob = Optimization('A trigonometric function', objfunc)
opt_prob.addVar('x1', 'c', lower=-10, upper=10, value=0.0)
opt_prob.addVar('x2', 'c', lower=-10, upper=10, value=0.0)
opt_prob.addObj('f')
opt_prob.addCon('g1', 'i')
opt_prob.addCon('g2', 'i')
opt_prob.addCon('g3', 'i')
# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

Solutions are:

```
Solution of a Semidefinite Program:
                Solver: CSDP
                Status: Optimal
   Initialization Time: 3.22915506363 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])

ALPSO Solution to A trigonometric function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                   0.3503
    Total Function Evaluations:     3640
    Lambda: [ 0.          0.         2.0077542]
    Seed: 1482111691.32805490

    Objectives:
        Name        Value        Optimum
             f        -11.8237             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1      c      7.854321      -1.00e+01     1.00e+01
             x2      c      4.549489      -1.00e+01     1.00e+01

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1          i       -1.00e+21 <= -10.000000 <= 0.00e+00
             g2          i       -1.00e+21 <= -8.649336 <= 0.00e+00
             g3          i       -1.00e+21 <= -0.000612 <= 0.00e+00

--------------------------------------------------------------------------------


NSGA-II Solution to A trigonometric function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                   0.7216
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f             -12             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1      c     -7.854036      -1.00e+01     1.00e+01
             x2      c      0.000004      -1.00e+01     1.00e+01

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1          i       -1.00e+21 <= -6.000000 <= 0.00e+00
```

```
          g2          i       -1.00e+21 <= -6.000000 <= 0.00e+00
          g3          i       -1.00e+21 <= -6.000000 <= 0.00e+00

--------------------------------------------------------------------------------
```

3.5 SOS Decomposition

Let f_* be the result of `SDPRelaxations.Minimize()`, then $f - f_* \in Q_{\mathbf{g}}$. Therefore, there exist $\sigma_0, \sigma_1, \ldots, \sigma_m \in \sum A^2$ such that $f - f_* = \sigma_0 + \sum_{i=1}^m \sigma_i g_i$. Once the `Minimize()` is called, the method `SDPRelaxations.Decompose()` returns this a dictionary of elements of A of the form `{0:[a(0, 1), ..., a(0, k_0)], ..., m:[a(m, 1), ..., a(m, k_m)}` such that

$$f - f_* = \sum_{i=0}^{m} g_i \sum_{j=1}^{k_i} a_{ij}^2,$$

where $g_0 = 1$.

Usually there are extra coefficients that are very small in absolute value as a result of round off error that should be ignored.

The following example shows how to employ this functionality:

```python
from sympy import *
from Irene import SDPRelaxations
# define the symbolic variables and functions
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')

Rlx = SDPRelaxations([x, y, z])
Rlx.SetObjective(x**3 + x**2 * y**2 + z**2 * x * y - x * z)
Rlx.AddConstraint(9 - (x**2 + y**2 + z**2) >= 0)
# initiate the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
# extract decomposition
V = Rlx.Decompose()
# test the decomposition
sos = 0
for v in V:
    # for g0 = 1
    if v == 0:
        sos = expand(Rlx.ReduceExp(sum([p**2 for p in V[v]])))
    # for g1, the constraint
    else:
        sos = expand(Rlx.ReduceExp(
            sos + Rlx.Constraints[v - 1] * sum([p**2 for p in V[v]])))
sos = sos.subs(Rlx.RevSymDict)
pln = Poly(sos).as_dict()
pln = {ex:round(pln[ex],5) for ex in pln}
print Poly(pln, (x,y,z)).as_expr()
```

The output looks like this:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 0.875229120255 seconds
              Run Time: 0.031426 seconds
Primal Objective Value: -27.4974076889
  Dual Objective Value: -27.4974076213
Feasible solution for moments of order 2

1.0*x**3 + 1.0*x**2*y**2 + 1.0*x*y*z**2 - 1.0*x*z + 27.49741
```

3.6 The `SDRelaxSol`

This object is a container for the solution of `SDPRelaxation` objects. It contains the following informations:

- *Primal*: the value of the SDP in primal form,
- *Dual*: the value of the SDP in dual form,
- *RunTime*: the run time of the sdp solver,
- *InitTime*: the total time consumed for initialization of the sdp,
- *Solver*: the name of sdp solver,
- *Status*: final status of the sdp solver,
- *RelaxationOrd*: order of relaxation,
- *TruncatedMmntSeq*: a dictionary of resulted moments,
- *MomentMatrix*: the resulted moment matrix,
- *ScipySolver*: the scipy solver to extract solutions,
- *err_tol*: the minimum value which is considered to be nonzero,
- *Support*: the support of discrete measure resulted from `SDPRelaxation.Minimize()`,
- *Weights*: corresponding weights for the Dirac measures.

3.6.1 Extracting solutions

By default, the support of the measure is not calculated, but it can be approximated by calling the method `SDRelaxSol.ExtractSolution()`.

There exists an exact theoretical method for extracting the support of the solution measure as explained in *[HL]*. But because of the numerical error of sdp solvers, computing rank and hence the support is quite difficult. So, `SDRelaxSol.ExtractSolution()` estimates the rank numerically by assuming that eigenvalues with absolute value less than `err_tol` which by default is set to `SDPRelaxation.ErrorTolerance`.

Two methods are implemented for extracting solutions:

- **Lasserre-Henrion** method as explained in *[HL]*. To employ this method simply call `SDRelaxSol.ExtractSolution('LH', card)`, where `card` is the maximum cardinality of the support.
- **Moment Matching** method which employs `scipy.optimize.root` to approximate the support. The default `scipy` solver is set to *lm*, but other solvers can be selected using `SDRelaxSol.SetScipySolver(solver)`. It is not guaranteed that scipy solvers return a reliable answer, but modifying

sdp solvers and other parameters like `SDPRelaxation.ErrorTolerance` may help to get better results. To use this method call `SDRelaxSol.ExtractSolution('scipy', card)` where `card` is as above.

Example 1. Solve and find minimizers of $x^2 + y^2 + z^4$ where $x + y + z = 4$:

```
from sympy import *
from Irene import *

x, y, z = symbols('x,y,z')

Rlx = SDPRelaxations([x, y, z])
Rlx.SetSDPSolver('cvxopt')
Rlx.SetObjective(x**2 + y**2 + z**4)
Rlx.AddConstraint(Eq(x + y + z, 4))
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
# extract support
Rlx.Solution.ExtractSolution('LH', 1)
print Rlx.Solution

# pyOpt
from pyOpt import *

def objfunc(x):
        f = x[0]**2 + x[1]**2 + x[2]**4
        g = [x[0] + x[1] + x[2] - 4]
        fail = 0
        return f, g, fail

opt_prob = Optimization('Testing solutions', objfunc)
opt_prob.addVar('x1', 'c', lower=-4, upper=4, value=0.0)
opt_prob.addVar('x2', 'c', lower=-4, upper=4, value=0.0)
opt_prob.addVar('x3', 'c', lower=-4, upper=4, value=0.0)
opt_prob.addObj('f')
opt_prob.addCon('g1', 'e')
# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
```

The output is:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 1.59334087372 seconds
              Run Time: 0.021102 seconds
Primal Objective Value: 5.45953579912
  Dual Objective Value: 5.45953586121
               Support:
                (0.91685039306810523, 1.541574317520042, 1.5415743175200163)
        Support solver: Lasserre--Henrion
Feasible solution for moments of order 2

ALPSO Solution to Testing solutions
================================================================================

        Objective Function: objfunc
```

```
    Solution:
--------------------------------------------------------------------------------
    Total Time:                      0.1443
    Total Function Evaluations:        1720
    Lambda: [-3.09182651]
    Seed: 1482274189.55335808

    Objectives:
        Name         Value         Optimum
             f          5.46051             0

        Variables (c - continuous, i - integer, d - discrete):
        Name     Type        Value      Lower Bound  Upper Bound
             x1       c       1.542371      -4.00e+00     4.00e+00
             x2       c       1.541094      -4.00e+00     4.00e+00
             x3       c       0.916848      -4.00e+00     4.00e+00

        Constraints (i - inequality, e - equality):
        Name     Type                    Bounds
             g1       e                 0.000314 = 0.00e+00

--------------------------------------------------------------------------------
```

Example 2. Minimize $-(x-1)^2-(x-y)^2-(y-3)^2$ where $1-(x-1)^2 \geq 0$, $1-(x-y)^2 \geq 0$ and $1-(y-3)^2 \geq 0$. It has three minimizers $(2,3)$, $(1,2)$, and $(2,2)$:

```
from sympy import *
from Irene import *

x, y = symbols('x, y')

Rlx = SDPRelaxations([x, y])
Rlx.SetSDPSolver('csdp')
Rlx.SetObjective(-(x - 1)**2 - (x - y)**2 - (y - 3)**2)
Rlx.AddConstraint(1 - (x - 1)**2 >= 0)
Rlx.AddConstraint(1 - (x - y)**2 >= 0)
Rlx.AddConstraint(1 - (y - 3)**2 >= 0)
Rlx.MomentsOrd(2)
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
# extract support
Rlx.Solution.ExtractSolution('LH')
print Rlx.Solution

# pyOpt
from pyOpt import *


def objfunc(x):
    f = -(x[0] - 1)**2 - (x[0] - x[1])**2 - (x[1] - 3)**2
    g = [
        (x[0] - 1)**2 - 1,
        (x[0] - x[1])**2 - 1,
        (x[1] - 3)**2 - 1
    ]
    fail = 0
```

```
    return f, g, fail

opt_prob = Optimization("Lasserre's Example", objfunc)
opt_prob.addVar('x1', 'c', lower=-3, upper=3, value=0.0)
opt_prob.addVar('x2', 'c', lower=-3, upper=3, value=0.0)
opt_prob.addObj('f')
opt_prob.addCon('g1', 'i')
opt_prob.addCon('g2', 'i')
opt_prob.addCon('g3', 'i')
# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

which results in:

```
Solution of a Semidefinite Program:
                Solver: CSDP
                Status: Optimal
   Initialization Time: 0.861004114151 seconds
              Run Time: 0.00645 seconds
Primal Objective Value: -2.0
  Dual Objective Value: -2.0
               Support:
                (2.000000006497352, 3.000000045123556)
                (0.9999999382958613l, 1.9999999487412694)
                (1.9999999970209055, 1.9999999029899564)
        Support solver: Lasserre--Henrion
Feasible solution for moments of order 2


ALPSO Solution to Lasserre's Example
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                    0.1353
    Total Function Evaluations:      1720
    Lambda: [ 0.08278879  0.08220848  0.        ]
    Seed: 1482307696.27431393

    Objectives:
        Name        Value        Optimum
             f               -2             0

        Variables (c - continuous, i - integer, d - discrete):
        Name     Type       Value       Lower Bound  Upper Bound
             x1       c       1.999967      -3.00e+00     3.00e+00
             x2       c       3.000000      -3.00e+00     3.00e+00

        Constraints (i - inequality, e - equality):
        Name     Type                    Bounds
```

```
            g1             i       -1.00e+21 <= -0.000065 <= 0.00e+00
            g2             i       -1.00e+21 <= 0.000065 <= 0.00e+00
            g3             i       -1.00e+21 <= -1.000000 <= 0.00e+00

--------------------------------------------------------------------------------


NSGA-II Solution to Lasserre's Example
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                        0.2406
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f        -1.99941             0

        Variables (c - continuous, i - integer, d - discrete):
        Name     Type        Value       Lower Bound  Upper Bound
             x1       c       1.999947      -3.00e+00     3.00e+00
             x2       c       2.000243      -3.00e+00     3.00e+00

        Constraints (i - inequality, e - equality):
        Name     Type                    Bounds
             g1            i       -1.00e+21 <= -0.000106 <= 0.00e+00
             g2            i       -1.00e+21 <= -1.000000 <= 0.00e+00
             g3            i       -1.00e+21 <= -0.000486 <= 0.00e+00

--------------------------------------------------------------------------------
```

Irene detects all minimizers correctly, but each *pyOpt* solvers only detect one. Note that we did not specify number of solutions, but the solver extracted them all.

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.

4.1 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.

4.1.1 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`, `pyOpt.ALPSO` and `pyOpt.NSGA2`:

```
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

# pyOpt
from pyOpt import *


def objfunc(x):
    from numpy import sin, exp, pi
    f = x[0] + exp(x[0] * sin(x[0]))
    g = [
        x[0]**2 - pi**2
    ]
    fail = 0
    return f, g, fail

opt_prob = Optimization('A mixed function', objfunc)
opt_prob.addVar('x1', 'c', lower=-pi, upper=pi, value=0.0)
opt_prob.addObj('f')
opt_prob.addCon('g1', 'i')
# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

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.        ])

ALPSO Solution to A mixed function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                    0.0683
    Total Function Evaluations:      1240
    Lambda:     [ 0.]
    Seed: 1482112089.31088901

    Objectives:
        Name        Value        Optimum
             f        -2.14159             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1       c      -3.141593      -3.14e+00     3.14e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1           i       -1.00e+21 <= 0.000000 <= 0.00e+00

--------------------------------------------------------------------------------


NSGA-II Solution to A mixed function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                    0.4231
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f        -2.14159             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1       c      -3.141593      -3.14e+00     3.14e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1           i       -1.00e+21 <= 0.000000 <= 0.00e+00
```

```
--------------------------------------------------------------------------------
```

Now instead of Taylor expansion, we use Legendre polynomials to estimate $e^{x\sin x}$. To find Legendre estimators, we use pyProximation which implements general Hilbert space methods (see Appendix-*pyProximation*):

```
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`.

4.1.2 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_f to Orth_f and B_g to Orth_g
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
```

```
# pyOpt
from pyOpt import *


def objfunc(x):
    from numpy import sin, sinh, exp, pi
    f = x[0] * sinh(x[1]) + exp(x[1] * sin(x[0]))
    g = [
        x[0]**2 - pi**2,
        x[1]**2 - pi**2
    ]
    fail = 0
    return f, g, fail

opt_prob = Optimization(
    'A trigonometric-hyperbolic-exponential function', objfunc)
opt_prob.addVar('x1', 'c', lower=-pi, upper=pi, value=0.0)
opt_prob.addVar('x2', 'c', lower=-pi, upper=pi, value=0.0)
opt_prob.addObj('f')
opt_prob.addCon('g1', 'i')
opt_prob.addCon('g2', 'i')
# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

The result will be:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 4.09241986275 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.])

ALPSO Solution to A trigonometric-hyperbolic-exponential function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                       0.0946
    Total Function Evaluations:         1240
    Lambda: [ 0.  0.]
    Seed: 1482112613.82665610

    Objectives:
        Name        Value        Optimum
             f        -35.2814              0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type        Value       Lower Bound  Upper Bound
             x1       c      -3.141593      -3.14e+00     3.14e+00
             x2       c       3.141593      -3.14e+00     3.14e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1           i       -1.00e+21 <= 0.000000 <= 0.00e+00
             g2           i       -1.00e+21 <= 0.000000 <= 0.00e+00

--------------------------------------------------------------------------------


NSGA-II Solution to A trigonometric-hyperbolic-exponential function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                       0.5331
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f        -35.2814              0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type        Value       Lower Bound  Upper Bound
             x1       c       3.141593      -3.14e+00     3.14e+00
             x2       c      -3.141593      -3.14e+00     3.14e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1           i       -1.00e+21 <= 0.000000 <= 0.00e+00
             g2           i       -1.00e+21 <= 0.000000 <= 0.00e+00

--------------------------------------------------------------------------------
```

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]*.

5.1 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:

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

Since f is a sum of squares, and $f(1, \dots, 1) = 0$, 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 $i = 1, \dots, 9$:

```
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)])
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

# pyOpt
from pyOpt import *


def objfunc(x):
    f = sum([100 * (x[i + 1] - x[i]**2)**2 + (1 - x[i])
             ** 2 for i in range(NumVars - 1)])
    g = [x[i]**2 - 9 for i in range(NumVars)]
    fail = 0
    return f, g, fail

opt_prob = Optimization(
    'The Rosenbrock function', objfunc)
opt_prob.addObj('f')
for i in range(NumVars):
    opt_prob.addVar('x%d' % (i + 1), 'c', lower=-3, upper=3, value=0.0)
    opt_prob.addCon('g%d' % (i + 1), 'i')

# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

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])

ALPSO Solution to The Rosenbrock function
================================================================================


        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                    10.6371
    Total Function Evaluations:      48040
    Lambda: [ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
    Seed: 1482114864.60097694

    Objectives:
        Name        Value        Optimum
             f        0.590722             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1       c       0.992774      -3.00e+00     3.00e+00
             x2       c       0.986019      -3.00e+00     3.00e+00
             x3       c       0.970756      -3.00e+00     3.00e+00
             x4       c       0.942489      -3.00e+00     3.00e+00
             x5       c       0.886910      -3.00e+00     3.00e+00
             x6       c       0.787367      -3.00e+00     3.00e+00
             x7       c       0.618875      -3.00e+00     3.00e+00
             x8       c       0.382054      -3.00e+00     3.00e+00
             x9       c       0.143717      -3.00e+00     3.00e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1           i       -1.00e+21 <= -8.014399 <= 0.00e+00
             g2           i       -1.00e+21 <= -8.027767 <= 0.00e+00
             g3           i       -1.00e+21 <= -8.057633 <= 0.00e+00
             g4           i       -1.00e+21 <= -8.111714 <= 0.00e+00
             g5           i       -1.00e+21 <= -8.213391 <= 0.00e+00
             g6           i       -1.00e+21 <= -8.380053 <= 0.00e+00
             g7           i       -1.00e+21 <= -8.616994 <= 0.00e+00
             g8           i       -1.00e+21 <= -8.854035 <= 0.00e+00
             g9           i       -1.00e+21 <= -8.979345 <= 0.00e+00
```

```
--------------------------------------------------------------------------------


NSGA-II Solution to The Rosenbrock function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                        0.6244
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f           5.5654             0

        Variables (c - continuous, i - integer, d - discrete):
        Name     Type        Value       Lower Bound  Upper Bound
             x1       c       0.727524      -3.00e+00     3.00e+00
             x2       c       0.537067      -3.00e+00     3.00e+00
             x3       c       0.296186      -3.00e+00     3.00e+00
             x4       c       0.094420      -3.00e+00     3.00e+00
             x5       c       0.017348      -3.00e+00     3.00e+00
             x6       c       0.009658      -3.00e+00     3.00e+00
             x7       c       0.015372      -3.00e+00     3.00e+00
             x8       c       0.009712      -3.00e+00     3.00e+00
             x9       c       0.001387      -3.00e+00     3.00e+00

        Constraints (i - inequality, e - equality):
        Name     Type                    Bounds
             g1           i       -1.00e+21 <= -8.470708 <= 0.00e+00
             g2           i       -1.00e+21 <= -8.711559 <= 0.00e+00
             g3           i       -1.00e+21 <= -8.912274 <= 0.00e+00
             g4           i       -1.00e+21 <= -8.991085 <= 0.00e+00
             g5           i       -1.00e+21 <= -8.999699 <= 0.00e+00
             g6           i       -1.00e+21 <= -8.999907 <= 0.00e+00
             g7           i       -1.00e+21 <= -8.999764 <= 0.00e+00
             g8           i       -1.00e+21 <= -8.999906 <= 0.00e+00
             g9           i       -1.00e+21 <= -8.999998 <= 0.00e+00

--------------------------------------------------------------------------------
```

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

5.2 Giunta Function

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

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

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

# pyOpt
from pyOpt import *


def objfunc(x):
    f = .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)))
    g = [x[i]**2 - 1 for i in range(2)]
    fail = 0
    return f, g, fail

opt_prob = Optimization(
    'The Giunta function', objfunc)
opt_prob.addObj('f')
for i in range(2):
    opt_prob.addVar('x%d' % (i + 1), 'c', lower=-1, upper=1, value=0.0)
    opt_prob.addCon('g%d' % (i + 1), 'i')

# Augmented Lagrangian Particle Swarm Optimizer
```

```
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

and the result is:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 2.53814482689 seconds
              Run Time: 0.041321 seconds
Primal Objective Value: 0.0644704534329
  Dual Objective Value: 0.0644704595475
Feasible solution for moments of order 2

solution according to 'COBYLA':
     fun: 0.064470430891900576
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 40
  status: 1
 success: True
       x: array([ 0.46730658,  0.4674184 ])
solution according to 'SLSQP':
     fun: 0.0644704633430450
     jac: array([-0.00029983, -0.00029983,  0.         ])
 message: 'Optimization terminated successfully.'
    nfev: 13
     nit: 3
    njev: 3
  status: 0
 success: True
       x: array([ 0.46717727,  0.46717727])

ALPSO Solution to The Giunta function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                    10.6180
    Total Function Evaluations:       1240
    Lambda: [ 0.  0.]
    Seed: 1482115204.08583212

    Objectives:
        Name        Value        Optimum
             f       0.0644704             0

        Variables (c - continuous, i - integer, d - discrete):
        Name     Type       Value       Lower Bound  Upper Bound
             x1       c       0.467346       -1.00e+00     1.00e+00
             x2       c       0.467369       -1.00e+00     1.00e+00
```

```
        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1            i       -1.00e+21 <= -0.781588 <= 0.00e+00
             g2            i       -1.00e+21 <= -0.781566 <= 0.00e+00

--------------------------------------------------------------------------------


NSGA-II Solution to The Giunta function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                      50.9196
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f       0.0644704             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1       c       0.467403      -1.00e+00     1.00e+00
             x2       c       0.467324      -1.00e+00     1.00e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1            i       -1.00e+21 <= -0.781535 <= 0.00e+00
             g2            i       -1.00e+21 <= -0.781608 <= 0.00e+00

--------------------------------------------------------------------------------
```

5.3 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

# pyOpt
from pyOpt import *


def objfunc(x):
    f = cos(x[0])**2 + sin(x[1])**2
    g = [x[i]**2 - 25 for i in range(2)]
    fail = 0
    return f, g, fail

opt_prob = Optimization(
    'The Parsopoulos function', objfunc)
opt_prob.addObj('f')
for i in range(2):
    opt_prob.addVar('x%d' % (i + 1), 'c', lower=-5, upper=5, value=0.0)
    opt_prob.addCon('g%d' % (i + 1), 'i')

# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

which returns:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 2.48692297935 seconds
              Run Time: 0.035358 seconds
Primal Objective Value: -3.74719295193e-10
  Dual Objective Value: 5.43053240402e-12
Feasible solution for moments of order 2

solution according to 'COBYLA':
     fun: 1.83716742579312e-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.])

ALPSO Solution to The Parsopoulos function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                      4.4576
    Total Function Evaluations:        1240
    Lambda: [ 0.  0.]
    Seed: 1482115438.17070389

    Objectives:
        Name        Value        Optimum
             f     5.68622e-09             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1      c      -4.712408      -5.00e+00     5.00e+00
             x2      c      -0.000073      -5.00e+00     5.00e+00

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1          i       -1.00e+21 <= -2.793212 <= 0.00e+00
             g2          i       -1.00e+21 <= -25.000000 <= 0.00e+00

--------------------------------------------------------------------------------


NSGA-II Solution to The Parsopoulos function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                     17.7197
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f     2.37167e-08             0
```

```
        Variables (c - continuous, i - integer, d - discrete):
        Name     Type        Value        Lower Bound  Upper Bound
             x1       c      -1.570676      -5.00e+00     5.00e+00
             x2       c       3.141496      -5.00e+00     5.00e+00

        Constraints (i - inequality, e - equality):
        Name     Type                    Bounds
             g1           i       -1.00e+21 <= -22.532977 <= 0.00e+00
             g2           i       -1.00e+21 <= -15.131000 <= 0.00e+00

--------------------------------------------------------------------------------
```

5.4 Shubert Function

Shubert function is defined by:

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

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)

# pyOpt
from pyOpt import *


def objfunc(x):
    f = 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)])
    g = [x[i]**2 - 100 for i in range(2)]
    fail = 0
    return f, g, fail

opt_prob = Optimization(
    'The Shubert function', objfunc)
opt_prob.addObj('f')
for i in range(2):
    opt_prob.addVar('x%d' % (i + 1), 'c', lower=-10, upper=10, value=0.0)
    opt_prob.addCon('g%d' % (i + 1), 'i')

# Augmented Lagrangian Particle Swarm Optimizer
alpso = ALPSO()
alpso(opt_prob)
print opt_prob.solution(0)
# Non Sorting Genetic Algorithm II
nsg2 = NSGA2()
nsg2(opt_prob)
print opt_prob.solution(1)
```

The result is:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 730.02412415 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

ALPSO Solution to The Shubert function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                     37.7526
    Total Function Evaluations:        2200
    Lambda: [ 0.  0.]
    Seed: 1482115770.57303905

    Objectives:
        Name        Value        Optimum
             f        -18.0956             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1       c      -7.061398      -1.00e+01     1.00e+01
             x2       c      -1.471424      -1.00e+01     1.00e+01

        Constraints (i - inequality, e - equality):
        Name    Type                    Bounds
             g1           i       -1.00e+21 <= -50.136654 <= 0.00e+00
             g2           i       -1.00e+21 <= -97.834910 <= 0.00e+00

--------------------------------------------------------------------------------


NSGA-II Solution to The Shubert function
================================================================================

        Objective Function: objfunc

    Solution:
--------------------------------------------------------------------------------
    Total Time:                     97.6291
    Total Function Evaluations:

    Objectives:
        Name        Value        Optimum
             f        -18.0955             0

        Variables (c - continuous, i - integer, d - discrete):
        Name    Type       Value       Lower Bound  Upper Bound
             x1       c      -0.778010      -1.00e+01     1.00e+01
```

```
            x2       c      -7.754277      -1.00e+01     1.00e+01

        Constraints (i - inequality, e - equality):
        Name     Type                    Bounds
             g1           i       -1.00e+21 <= -99.394700 <= 0.00e+00
             g2           i       -1.00e+21 <= -39.871193 <= 0.00e+00

--------------------------------------------------------------------------------
```

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

Moreover, we employed 20 different optimizers included in pyOpt and only 4 of them returned the correct optimum value.

5.5 McCormick Function

McCormick function is defined by

$$f(x, y) = \sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1.$$

Attains its minimum at $f(-.54719, -1.54719) \approx -1.9133$:

```
from sympy import *
from Irene import *
from pyProximation import OrthSystem
# introduce symbols
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
# transcendental term of objective
f = sin(z)
# Legendre polynomials via pyProximation
D_f = [(-2, 2)]
Orth_f = OrthSystem([z], D_f)
# set bases
B_f = Orth_f.PolyBasis(10)
# link B_f to Orth_f
Orth_f.Basis(B_f)
# generate the orthonormal bases
Orth_f.FormBasis()
# extract the coefficients of approximations
Coeffs_f = Orth_f.Series(f)
# form the approximations
f_app = sum([Orth_f.OrthBase[i] * Coeffs_f[i]
             for i in range(len(Orth_f.OrthBase))])
# objective function
obj = f_app.subs({z: x + y}) + (x - y)**2 - 1.5 * x + 2.5 * y + 1
# initiate the Relaxation object
Rlx = SDPRelaxations([x, y])
# set the objective
Rlx.SetObjective(obj)
# add support constraints
Rlx.AddConstraint(4 - (x**2 + y**2) >= 0)
# set the sdp solver
Rlx.SetSDPSolver('cvxopt')
# initialize the SDP
```

```
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
Rlx.Solution.ExtractSolution('lh',1)
print Rlx.Solution
```

Results in:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 10.6071600914 seconds
              Run Time: 0.070002 seconds
Primal Objective Value: -1.91322353633
  Dual Objective Value: -1.91322352558
               Support:
                (-0.54724056855672309, -1.5473099043318805)
        Support solver: Lasserre--Henrion
Feasible solution for moments of order 5
```

5.6 Schaffer Function N.2

Schaffer function N.2 is

$$f(x, y) = \frac{\sin^2(x^2 - y^2) - .5}{(1 + .001(x^2 + y^2))^2}.$$

Attains its minimum at $f(0, 0) = .5$:

```
from sympy import *
from Irene import *
from pyProximation import OrthSystem, Measure
# introduce symbols and functions
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
# transcendental term of objective
f = (sin(z))**2
# Chebyshev polynomials via pyProximation
D_f = [(-2, 2)]
w = lambda x: 1. / sqrt(4 - x**2)
M = Measure(D_f, w)
# link the measure to S
Orth_f = OrthSystem([z], D_f)
Orth_f.SetMeasure(M)
# set bases
B_f = Orth_f.PolyBasis(8)
# link B to S
Orth_f.Basis(B_f)
# generate the orthonormal bases
Orth_f.FormBasis()
# extract the coefficients of approximations
Coeffs_f = Orth_f.Series(f)
# form the approximations
f_app = sum([Orth_f.OrthBase[i] * Coeffs_f[i]
             for i in range(len(Orth_f.OrthBase))])
```

```
# objective function
obj = f_app.subs({z: x**2 - y**2}) - .5
# initiate the Relaxation object
Rlx = SDPRelaxations([x, y])
# settings
Rlx.Probability = False
# set the objective
Rlx.SetObjective(obj)
# add support constraints
Rlx.AddConstraint(4 - (x**2 + y**2) >= 0)
# moment constraint
Rlx.MomentConstraint(Mom((1 + .001 * (x**2 + y**2)**2)) == 1)
# set the sdp solver
Rlx.SetSDPSolver('cvxopt')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
Rlx.Solution.ExtractSolution('lh', 1)
print Rlx.Solution
```

The result:

```
Solution of a Semidefinite Program:
                Solver: CVXOPT
                Status: Optimal
   Initialization Time: 26.6285181046 seconds
              Run Time: 0.110288 seconds
Primal Objective Value: -0.495770329702
  Dual Objective Value: -0.495770335895
               Support:
                (1.3348173524856991e-15, 8.3700760032311997e-17)
        Support solver: Lasserre--Henrion
Feasible solution for moments of order 6
```

5.7 Schaffer Function N.4

Schaffer function N.4 is

$$f(x,y) = \frac{\cos^2(\sin(|(x^2-y^2)|)) - .5}{(1+.001(x^2+y^2))^2}.$$

The minimum value is -0.207421:

```
from sympy import *
from Irene import *
from pyProximation import OrthSystem, Measure
# introduce symbols and functions
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
# transcendental term of objective
f = (cos(sin(abs(z))))**2
# Chebyshev polynomials via pyProximation
D_f = [(-2, 2)]
w = lambda x: 1. / sqrt(4 - x**2)
```

```
M = Measure(D_f, w)
# link the measure to S
Orth_f = OrthSystem([z], D_f)
Orth_f.SetMeasure(M)
# set bases
B_f = Orth_f.PolyBasis(12)
# link B_f to Orth_f
Orth_f.Basis(B_f)
# generate the orthonormal bases
Orth_f.FormBasis()
# extract the coefficients of approximations
Coeffs_f = Orth_f.Series(f)
# form the approximations
f_app = sum([Orth_f.OrthBase[i] * Coeffs_f[i]
             for i in range(len(Orth_f.OrthBase))])
# objective function
obj = f_app.subs({z: x**2 - y**2}) - .5
# initiate the Relaxation object
Rlx = SDPRelaxations([x, y])
# settings
Rlx.Probability = False
# set the objective
Rlx.SetObjective(obj)
# add support constraints
Rlx.AddConstraint(4 - (x**2 + y**2) >= 0)
# moment constraint
Rlx.MomentConstraint(Mom((1 + .001 * (x**2 + y**2)**2)) == 1)
# set the sdp solver
Rlx.SetSDPSolver('csdp')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
print Rlx.Solution
```

Result is:

```
Solution of a Semidefinite Program:
                Solver: DSDP
                Status: Optimal
   Initialization Time: 497.670987129 seconds
              Run Time: 75.423031 seconds
Primal Objective Value: -0.203973186683
  Dual Objective Value: -0.208094722977
Feasible solution for moments of order 12
```

5.8 Drop-Wave Function

The Drop-Wave function is multimodal and highly complex:

$$f(x, y) = -\frac{1 + \cos(12\sqrt{x^2 + y^2})}{.5(x^2 + y^2) + 2}$$

It has a global minimum at $f(0, 0) = -1$:

```python
from sympy import *
from Irene import *
from pyProximation import OrthSystem, Measure
# introduce symbols and functions
x = Symbol('x')
y = Symbol('y')
z = Symbol('z')
s = Symbol('s')
# transcendental term of objective
f = cos(z)
# Legendre polynomials via pyProximation
D_f = [(-2, 2)]
# link the measure to S
Orth_f = OrthSystem([z], D_f)
# set bases
B_f = Orth_f.PolyBasis(8)
# link B_f to Orth_f
Orth_f.Basis(B_f)
# generate the orthonormal bases
Orth_f.FormBasis()
# extract the coefficients of approximations
Coeffs_f = Orth_f.Series(f)
# form the approximations
f_app = sum([Orth_f.OrthBase[i] * Coeffs_f[i]
             for i in range(len(Orth_f.OrthBase))])
# objective function
obj = -1 - f_app.subs({z: 12 * s})
# relations
rels = [s**2 - (x**2 + y**2)]
# initiate the Relaxation object
Rlx = SDPRelaxations([x, y, s])
# settings
Rlx.Probability = False
# set the objective
Rlx.SetObjective(obj)
# add support constraints
Rlx.AddConstraint(4 - s**2 >= 0)
# moment constraint
Rlx.MomentConstraint(Mom((2 + .5 * s**2) == 1))
# set the sdp solver
Rlx.SetSDPSolver('dsdp')
# initialize the SDP
Rlx.InitSDP()
# solve the SDP
Rlx.Minimize()
Rlx.Solution.ExtractSolution('lh', 1)
print Rlx.Solution
```

The output is:

```
Solution of a Semidefinite Program:
                Solver: DSDP
                Status: Optimal
   Initialization Time: 24.8883750439 seconds
              Run Time: 1.85318 seconds
Primal Objective Value: -86177004058.9
  Dual Objective Value: -0.0
               Support:
```

```
               (5.015829856065089e-08, -1.5584184369059079e-09, -2.7794786040781493e-
↪09)
       Support solver: Lasserre--Henrion
Feasible solution for moments of order 4
```

Note that although the solver did not converge, but the dual objective value and calculated support are correct.

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.

`relaxations.`**`Calpha_`**`(`*expn*, *Mmnt*`)`

Given an exponent *expn*, this function finds the corresponding C_{expn} matrix which can be used for parallel processing.

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.

`Decompose``()`

Returns a dictionary that associates a list to every constraint, $g_i \geq 0$ for $i = 0, \ldots, m$, where $g_0 = 1$. Each list consists of elements of algebra whose sums of squares is equal to σ_i and $f - f_* = \sum_{i=0}^{m} \sigma_i g_i$. Here, f_* is the output of the `SDPRelaxation.Minimize()`.

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 SDP based on availability of `joblib`. If it is available, it runs in parallel mode, otherwise in serial.

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

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*.

SetNumCores(*num*)
Sets the maximum number of workers which cannot be bigger than number of available cores.

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.

getConstraint(*idx*)
Returns the constraint number *idx* of the problem after reduction modulo the relations, if given.

getMomentConstraint(*idx*)
Returns the moment constraint number *idx* of the problem after reduction modulo the relations, if given.

getObjective()
Returns the objective function of the problem after reduction modulo the relations, if given.

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

sInitSDP()
Initializes the semidefinite program (SDP), in serial mode, whose solution is a lower bound for the minimum of the program.

class `relaxations.`**SDRelaxSol**(*X*, *symdict={}*, *err_tol=1e-05*)
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
- `SDRelaxSol.ScipySolver` the scipy solver to extract solutions
- `SDRelaxSol.err_tol` the minimum value which is considered to be nonzero
- `SDRelaxSol.Support` the support of discrete measure resulted from `SDPRelaxation.Minimize()`
- `SDRelaxSol.Weights` corresponding weights for the Dirac measures

ExtractSolution(*mthd='LH'*, *card=0*)
Extract support of the solution measure from `SDPRelaxations`:

-`mthd` should be either 'LH' or 'Scipy', where 'LH' stands for 'Lasserre-Henrion' and 'Scipy' employs a Scipy solver to find points matching the moments,

-`card` restricts the number of points of the support.

ExtractSolutionLH(*card=0*)
Extract solutions based on Lasserre–Henrion's method.

ExtractSolutionScipy(*card=0*)
This method tries to extract the corresponding values for generators of the `SDPRelaxation` class. Number of points is the rank of the moment matrix which is computed numerically according to the size of its eigenvalues. Then the points are extracted as solutions of a system of polynomial equations using a *scipy* solver. The followin solvers are currently acceptable by `scipy`:

- `hybr`,
- `lm` (default),
- `broyden1`,
- `broyden2`,
- `anderson`,
- `linearmixing`,
- `diagbroyden`,
- `excitingmixing`,
- `krylov`,
- `df-sane`.

NumericalRank()
Finds the rank of the moment matrix based on the size of its eigenvalues. It considers those with absolute value less than `self.err_tol` to be zero.

Pivot(*arr*)
Get the leading term of arr > tau

SetScipySolver(*solver*)
Sets the `scipy.optimize.root` solver to *solver*.

StblRedEch(*A*)
Compute the stabilized row reduced echelon form.

Term2Mmnt(*trm*, *rnk*, *X*)
Converts a moment object into an algebraic equation.

class `sdp.`**sdp**(*solver='cvxopt'*)
This is the class which intends to solve semidefinite programs in primal format:

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

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 $C = [C_1, \dots, C_k]$.

AddConstraintBlock(*A*)
This takes a list of square matrices which corresponds to coefficient of x_i. Simply, $A_i = [A_{i1}, \dots, A_{ik}]$. 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. The following options are supported by solvers:

- `CVXOPT`:
  - `show_progress`: `True` or `False`, turns the output to the screen on or off (default: `True`);
  - `maxiters`: maximum number of iterations (default: 100);
  - `abstol`: absolute accuracy (default: 1e-7);
  - `reltol`: relative accuracy (default: 1e-6);
  - `feastol`: tolerance for feasibility conditions (default: 1e-7);
  - `refinement`: number of iterative refinement steps when solving KKT equations (default: 0 if the problem has no second-order cone or matrix inequality constraints; 1 otherwise).
- `SDPA`:
  - `maxIteration`: Maximum number of iterations. The SDPA stops when the iteration exceeds `maxIteration`;
  - `epsilonStar`, `epsilonDash`: The accuracy of an approximate optimal solution of the SDP;
  - `lambdaStar`: This parameter determines an initial point;
  - `omegaStar`: This parameter determines the region in which the SDPA searches an optimal solution;
  - `lowerBound`: Lower bound of the minimum objective value of the primal problem;
  - `upperBound`: Upper bound of the maximum objective value of the dual problem;
  - `betaStar`: Parameter controlling the search direction when current state is feasible;
  - `betaBar`: Parameter controlling the search direction when current state is infeasible;
  - `gammaStar`: Reduction factor for the primal and dual step lengths; 0.0 < `gammaStar` < 1.0.

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.

sdpa_param()
Produces sdpa.param file from `SolverOptions`.

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 1.1.0 (December 25, 2016- Merry Christmas)

- Extracting minimizers via `SDRelaxSol.ExtractSolution()` and help of `scipy`,
- Extracting minimizers implementing Lasserre-Henrion algorithm,
- Adding `SDPRelaxations.Probability` and `SDPRelaxations.PSDMoment` to give more flexibility over moments and enables rational minimization.
- SOS decomposition implemented.
- `__str__` method for `SDPRelaxations`.
- Using *pyOpt* as the external optimizer.
- More benchmark examples.

Version 1.0.0 (Dec 07, 2016)

- Initial release (Irene's birthday)

CHAPTER

EIGHT

TO DO

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

- Reduce dependency on SymPy.
- Keep track of original expressions before reduction.
- Write a LaTeX method.
- Include sdp solvers installation (subject to copyright limitations).
- Error handling for CSDP and SDPA failure.

8.1 Done

The following to-dos were implemented:

- Extract solutions (at least for polynomials)- in v.1.1.0.
- SOS decomposition- in v.1.1.0.
- Write a `__str__` method for `SDPRelaxations` printing- in v.1.1.0.

CHAPTER NINE

APPENDIX

9.1 pyProximation

pyProximation is a python package that was originally developed to solve integro-differential equations based on approximation on Hilbert function spaces. Thus, it has basic functionalities for computations via measures, generating orthonormal systems of functions from a given basis, interpolation and collocation method as well as some graphics.

For the purpose of this package, we are mainly interested in finding reliable approximations of certain functions. This can be done via `pyProximation.OrthSystem`. The relevant documentation can be found here.

Suppose that we want to approximate a given function $f(x)$ with Chebyshev polynomials of a certain degree n. Chebyshev polynomials are elements of the orthonormal basis obtained from Gram-Schmidt process applied to a monomial basis where the inner product is defined by

$$\langle p, q\rangle = \int_{-1}^{1} p\cdot q\, d\mu.$$

In this case $d\mu = \frac{dx}{\sqrt{1-x^2}}$. The following code, first generate such an orthonormal basis and then extracts coefficients of the approximation and then the Chebyshev approximation:

```
from sympy import *
from numpy import sqrt
from pyProximation import Measure, OrthSystem
# the symbolic variable
x = Symbol('x')
# set a limit to the order
n = 6
# define the measure
D = [(-1, 1)]
w = lambda x: 1./sqrt(1. - x**2)
M = Measure(D, w)
S = OrthSystem([x], D, 'sympy')
# link the measure to S
S.SetMeasure(M)
# set B = {1, x, x^2, ..., x^n}
B = S.PolyBasis(n)
# link B to S
S.Basis(B)
# generate the orthonormal basis
S.FormBasis()
m = len(S.OrthBase)
# set f(x) = sin(x)e^x
f = sin(x)*exp(x)
# extract the coefficients
```

```
Coeffs = S.Series(f)
# form the approximation
f_aprx = sum([S.OrthBase[i]*Coeffs[i] for i in range(m)])
print f_aprx
```

9.2 pyOpt

pyOpt is a Python-based package for formulating and solving nonlinear constrained optimization problems in an efficient, reusable and portable manner. It is an open-source software distributed under the terms of the GNU Lesser General Public License.

***pyOpt* provides unified interface to the following nonlinear optimizers:**

- SNOPT - Sparse NOlinear OPTimizer
- NLPQL - Non-Linear Programming by Quadratic Lagrangian
- NLPQLP - NonLinear Programming with Non-Monotone and Distributed Line Search
- FSQP - Feasible Sequential Quadratic Programming
- SLSQP - Sequential Least Squares Programming
- PSQP - Preconditioned Sequential Quadratic Programming
- ALGENCAN - Augmented Lagrangian with GENCAN
- FILTERSD
- MMA - Method of Moving Asymptotes
- GCMMA - Globally Convergent Method of Moving Asymptotes
- CONMIN - CONstrained function MINimization
- MMFD - Modified Method of Feasible Directions
- KSOPT - Kreisselmeier–Steinhauser Optimizer
- COBYLA - Constrained Optimization BY Linear Approximation
- SDPEN - Sequential Penalty Derivative-free method for Nonlinear constrained optimization
- SOLVOPT - SOLver for local OPTimization problems
- ALPSO - Augmented Lagrangian Particle Swarm Optimizer
- NSGA2 - Non Sorting Genetic Algorithm II
- ALHSO - Augmented Lagrangian Harmony Search Optimizer
- MIDACO - Mixed Integer Distributed Ant Colony Optimization

9.2.1 Basic usage:

pyOpt is design to solve general constrained nonlinear optimization problems:

$$\left\{\begin{array}{lll}\min & f(x) & \\ \text{Subject to} & & \\ & g_j(x)=0 & j=1,\dots,m_e \\ & g_j(x)\leq 0 & j=m_e+1,\dots,m \\ & l_i\leq x_i\leq u_i & i=1,\dots,n,\end{array}\right.$$

where:

- x is the vector of design variables
- $f(x)$ is a nonlinear function
- $g(x)$ is a linear or nonlinear function
- n is the number of design variables
- m_e is the number of equality constraints
- m is the total number of constraints (number of equality constraints: $m_i = m - m_e$).

The following is a pseudo-code demonstrating the basic usage of `pyOpt`:

```
# General Objective Function Template:
def obj_fun(x, *args, **kwargs):
        """
        f: objective value
        g: array (or list) of constraint values
        fail: 0 for successful function evaluation, 1 for unsuccessful function
↪evaluation (test must be provided by user)
        If the Optimization problem is unconstrained, g must be returned as an empty
↪list or array: g = []
        Inequality constraints are handled as `<=`.
        """
        fail = 0
        f = function(x,*args,**kwargs)
        g = function(x,*args,**kwargs)

        return f,g,fail

# Instantiating an Optimization Problem:
opt_prob = Optimization('name', obj_fun)
# Assigning Objective:
opt_prob.addObj('name', value=0.0, optimum=0.0)
# Single Design variable:
opt_prob.addVar('name', type='c', value=0.0, lower=-inf, upper=inf,
↪choices=listochoices)
# A Group of Design Variables:
opt_prob.addVarGroup('name', numerinGroup, type='c', value=value, lower=lb, upper=up,
↪choices=listochoices)
# where `value`, `lb`, `ub` (float or int or list or 1Darray).
# and supported Types are 'c': continuous design variable;
# `i`: integer design variable;
# `d`: discrete design variable (based on choices, e.g.: list/dict of materials).
# Assigning Constraints:
## Single Constraint:
opt_prob.addCon('name', type='i', lower=-inf, upper=inf, equal=0.0)
## A Group of Constraints:
opt_prob.addConGroup('name', numberinGroup, type='i', lower=lb, upper=up, equal=eq)
# where `lb`, `ub`, `eq` are (float or int or list or 1Darray).
# and supported types are
# `i` - inequality constraint;
# `e` - equality constraint.

# Instantiating an Optimizer (e.g.: Snopt):
opt = pySNOPT.SNOPT()
# Solving the Optimization Problem:
opt(opt_prob, sens_type='FD', disp_opts=False, sens_mode='', *args, **kwargs)
```

```
# Output:
print opt_prob
```

For more details, see pyOpt documentation.

BIBLIOGRAPHY

[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).

[HL] D. Henrion and J-B. Lasserre, *Detecting Global Optimality and Extracting Solutions in GloptiPoly*, Positive Polynomials in Control, LNCIS 312, 293-310 (2005).

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

PYTHON MODULE INDEX

INDEX

T

V

W