# Introduction

## Common Perspective of Ellipsoid Method

* It is commonly believed that it is inefficient in practice for large-scale problems.
  + The convergent rate is slow, even with the use of deep cuts.
  + Cannot exploit sparsity.
* Since then, it was supplanted by interior-point methods.
* Only treated as a theoretical tool for proving the polynomial-time solvability of combinatorial optimization problems.

## But…

* The ellipsoid method works very differently compared with the interior point method.
* Only require a *separtion oracle*. Can play nicely with other techniques.
* The ellipsoid method itself cannot exploit sparsity, but the oracle can.

## Consider Ellipsoid Method When…

* The number of optimization variables is moderate, e.g. ECO flow, analog circuit sizing, parametric problems
* The number of constraints is large, or even infinite
* Oracle can be implemented efficiently.

class: middle, center

# Cutting-plane Method Revisited

## Basic Idea

.pull-left70[

* Let be a convex set.
* Consider the feasibility problem:
  + Find a point in ,
  + or determine that is empty (i.e., no feasible solution)

] .pull-right30[

img

img

]

## Separation Oracle

.pull-left70[

* When a separation oracle is *queried* at , it either
  + asserts that , or
  + returns a separating hyperplane between and :

] .pull-right30[

img

img

]

## Separation oracle (cont’d)

* is called a *cutting-plane*, or cut, since it eliminates the halfspace from our search.
* If ( is on the boundary of halfspace that is cut), cutting-plane is called *neutral cut*.
* If ( lies in the interior of halfspace that is cut), cutting-plane is called *deep cut*.
* If ( lies in the exterior of halfspace that is cut), cutting-plane is called *shallow cut*.

## Subgradient

* is usually given by a set of inequalities or for , where is a convex function.
* A vector is called a subgradient of a convex function at if .
* Hence, the cut is given by

Remarks:

* If is differentiable, we can simply take

## Key components of Cutting-plane method

* Cutting plane oracle
* A search space initially big enough to cover , e.g.
  + Polyhedron =
  + Interval = (for one-dimensional problem)
  + Ellipsoid =

## Generic Cutting-plane method

* **Given** initial known to contain .
* **Repeat**
  1. Choose a point in
  2. Query the cutting-plane oracle at
  3. **If** , quit
  4. **Else**, update to a smaller set that covers:
  5. **If** or it is small enough, quit.

## Corresponding Python code

.small[

def cutting\_plane\_feas(Omega, S, options=Options())  
 feasible = False  
 status = 0  
 for niter in range(options.max\_it):  
 cut = Omega(S.xc) # query the oracle at S.xc  
 if cut is None: # feasible sol'n obtained  
 feasible = True  
 break  
 status, tsq = S.update(cut) # update S  
 if status != 0:  
 break  
 if tsq < options.tol:  
 status = 2  
 break  
 return CInfo(feasible, niter + 1, status)

]

## From Feasibility to Optimization

* The optimization problem is treated as a feasibility problem with an additional constraint
* could be a convex function or a *quasiconvex function*.
* is also called the *best-so-far* value of .

## Convex Optimization Problem

* Consider the following general form:
* where is the -sublevel set of .
* 👉 Note: if and only if (monotonicity)
* One easy way to solve the optimization problem is to apply the binary search on .

.small[

def bsearch(Omega, I, options=Options()):  
 # assume monotone  
 lower, upper = I  
 u\_orig = upper  
 for niter in range(options.max\_it):  
 t = lower + (upper - lower) / 2  
 if Omega(t): # feasible sol'n obtained  
 upper = t  
 else:  
 lower = t  
 tau = (upper - lower) / 2  
 if tau < options.tol:  
 break  
 ret = CInfo(upper != u\_orig, niter + 1, 0)  
 ret.value = upper  
 return ret

]

.small[

class bsearch\_adaptor:  
 def \_\_init\_\_(self, P, S, options=Options()):  
 self.P = P  
 self.S = S  
 self.options = options  
  
 @property  
 def x\_best(self):  
 return self.S.xc  
  
 def \_\_call\_\_(self, t):  
 S = self.S.copy()  
 self.P.update(t)  
 ell\_info = cutting\_plane\_feas(self.P, S, self.options)  
 if ell\_info.feasible:  
 self.S.xc = S.xc  
 return True  
 return False

]

## Shrinking

* Another possible way is, to update the best-so-far whenever a feasible solution is found by solving the equation:
* If the equation is difficuit to solve but is also convex w.r.t. , then we may create a new varaible, say and let .

## Generic Cutting-plane method (Optim)

* **Given** initial known to contain .
* **Repeat**
  1. Choose a point in
  2. Query the separation oracle at
  3. **If** , update such that .
  4. Update to a smaller set that covers:
  5. **If** or it is small enough, quit.

.small[

def cutting\_plane\_dc(Omega, S, t, options=Options()):  
 x\_best = S.xc # this is copy  
 t\_orig = t  
  
 for niter in range(options.max\_it):  
 cut, t1 = Omega(S.xc, t)  
 if t != t1: # best t obtained  
 t = t1  
 x\_best = S.xc  
 status, tsq = S.update(cut)  
 if status != 0:  
 break  
 if tsq < options.tol:  
 status = 2  
 break  
  
 ret = CInfo(t != t\_orig, niter + 1, status)  
 ret.value = t  
 return x\_best, ret

]

## Example - Profit Maximization Problem

This example is taken from [@Aliabadi2013Robust].

* : Cobb-Douglas production function
* : the market price per unit
* : the scale of production
* : the output elasticities
* : input quantity
* : output price
* : a given constant that restricts the quantity of

## Example - Profit maximization (cont’d)

* The formulation is not in the convex form.
* Rewrite the problem in the following form:

## Profit maximization in Convex Form

* By taking the logarithm of each variable:
  + , .
* We have the problem in a convex form:

.small[

class profit\_oracle:  
 def \_\_init\_\_(self, params, a, v):  
 p, A, k = params  
 self.log\_pA = np.log(p \* A)  
 self.log\_k = np.log(k)  
 self.v = v  
 self.a = a  
  
 def \_\_call\_\_(self, y, t):  
 fj = y[0] - self.log\_k # constraint  
 if fj > 0.:  
 g = np.array([1., 0.])  
 return (g, fj), t  
 log\_Cobb = self.log\_pA + self.a @ y  
 x = np.exp(y)  
 vx = self.v @ x  
 te = t + vx  
 fj = np.log(te) - log\_Cobb  
 if fj < 0.:  
 te = np.exp(log\_Cobb)  
 t = te - vx  
 fj = 0.  
 g = (self.v \* x) / te - self.a  
 return (g, fj), t

]

.small[

# Main program  
  
import numpy as np  
from ellpy.cutting\_plane import cutting\_plane\_dc  
from ellpy.ell import ell  
from .profit\_oracle import profit\_oracle  
  
p, A, k = 20., 40., 30.5  
params = p, A, k  
alpha, beta = 0.1, 0.4  
v1, v2 = 10., 35.  
a = np.array([alpha, beta])  
v = np.array([v1, v2])  
y0 = np.array([0., 0.]) # initial x0  
r = np.array([100., 100.]) # initial ellipsoid (sphere)  
E = ell(r, y0)  
P = profit\_oracle(params, a, v)  
yb1, ell\_info = cutting\_plane\_dc(P, E, 0.)  
print(ell\_info.value, ell\_info.feasible)

]

## Area of Applications

* Robust convex optimization
  + oracle technique: affine arithmetic
* Parametric network potential problem
  + oracle technique: negative cycle detection
* Semidefinite programming
  + oracle technique: Cholesky or factorization

class: middle, center

# Robust Convex Optimization

## Robust Optimization Formulation

* Consider:
* where represents a set of varying parameters.
* The problem can be reformulated as:

## Example - Profit Maximization Problem (convex)

* Now assume that:
  + and vary and respectively.
  + , , , and all vary .

## Example - Profit Maximization Problem (oracle)

By detail analysis, the worst case happens when:

* ,
* , ,
* if , , else
* if , , else

.small[

class profit\_rb\_oracle:  
 def \_\_init\_\_(self, params, a, v, vparams):  
 e1, e2, e3, e4, e5 = vparams  
 self.a = a  
 self.e = [e1, e2]  
 p, A, k = params  
 params\_rb = p - e3, A, k - e4  
 self.P = profit\_oracle(params\_rb, a, v + e5)  
  
 def \_\_call\_\_(self, y, t):  
 a\_rb = self.a.copy()  
 for i in [0, 1]:  
 a\_rb[i] += self.e[i] if y[i] <= 0 else -self.e[i]  
 self.P.a = a\_rb  
 return self.P(y, t)

]

## Oracle in Robust Optimization Formulation

* The oracle only needs to determine:
  + If for some and , then
    - the cut =
  + If for some , then
    - the cut =
  + Otherwise, is feasible, then
    - Let .
    - .
    - The cut =

Remark:

* for more complicated problems, affine arithmetic could be used [@liu2007robust].

class: middle, center

# Multi-parameter Network Problem

## Parametric Network Problem

Given a network represented by a directed graph .

Consider:

* is the concave function of edge ,
* Assume: network is large but the number of parameters is small.

## Network Potential Problem (cont’d)

Given , the problem has a feasible solution if and only if contains no negative cycle. Let be a set of all cycles of .

* is a cycle of
* .

## Negative Cycle Finding

There are lots of methods to detect negative cycles in a weighted graph [@cherkassky1999negative], in which Tarjan’s algorithm [@Tarjan1981negcycle] is one of the fastest algorithms in practice [@alg:dasdan\_mcr; @cherkassky1999negative].

## Oracle in Network Potential Problem

* The oracle only needs to determine:
  + If there exists a negative cycle under , then
    - the cut =
  + Otherwise, the shortest path solution gives the value of .

## Python Code

.small[

class network\_oracle:  
 def \_\_init\_\_(self, G, dist, h):  
 self.G = G  
 self.dist = dist  
 self.h = h  
 self.S = negCycleFinder(G)  
  
 def update(self, t):  
 self.h.update(t)  
  
 def \_\_call\_\_(self, x):  
 def get\_weight(G, e):  
 return self.h.eval(G, e, x)  
  
 C = self.S.find\_neg\_cycle(self.dist, get\_weight)  
 if C is None:  
 return None  
 f = -sum(self.h.eval(self.G, e, x) for e in C)  
 g = -sum(self.h.grad(self.G, e, x) for e in C)  
 return g, f

]

## Example - Optimal Matrix Scaling [@orlin1985computing]

* Given a sparse matrix .
* Find another matrix where is a nonnegative diagonal matrix, such that the ratio of any two elements of in absolute value is as close to 1 as possible.
* Let . Under the min-max-ratio criterion, the problem can be formulated as:

## Optimal Matrix Scaling (cont’d)

By taking the logarithms of variables, the above problem can be transformed into:

where denotes and .

.small[

class optscaling\_oracle:  
 def \_\_init\_\_(self, G, dist):  
 class ratio:  
 def eval(self, G, e, x):  
 u, v = e  
 cost = G[u][v]['cost']  
 return x[0] - cost if u < v else cost - x[1]  
  
 def grad(self, G, e, x):  
 u, v = e  
 return np.array([1., 0.] if u < v else [0., -1.])  
  
 self.network = network\_oracle(G, dist, ratio())  
  
 def \_\_call\_\_(self, x, t):  
 cut = self.network(x)  
 if cut:  
 return cut, t  
 s = x[0] - x[1]  
 fj = s - t  
 if fj < 0.:  
 t = s  
 fj = 0.  
 return (np.array([1., -1.]), fj), t

]

## Example - clock period & yield-driven co-optimization

* 👉 Note that is not concave in general in .
* Fortunately, we are most likely interested in optimizing circuits for high yield rather than the low one in practice.
* Therefore, by imposing an additional constraint to , say , the problem becomes convex.

## Example - clock period & yield-driven co-optimization

The problem can be reformulated as:

class: middle, center

# Matrix Inequalities

## Problems With Matrix Inequalities

Consider the following problem:

* : a matrix-valued function
* denotes is positive semidefinite.

## Problems With Matrix Inequalities

* Recall that a matrix is positive semidefinite if and only if for all .
* The problem can be transformed into:
* Consider is concave for all w. r. t. , then the above problem is a convex programming.
* Reduce to *semidefinite programming* if is linear w.r.t. , i.e.,

## Oracle in Matrix Inequalities

The oracle only needs to:

* Perform a *row-based* LDLT factorization such that .
* Let denotes a submatrix .
* If the process fails at row ,
  + there exists a vector , such that
    - , and
    - .
  + The cut =

## Lazy evaluation

* Don’t construct the full matrix at each iteration!
* Only O() per iteration, independent of !

.small[

class lmi\_oracle:  
 ''' Oracle for LMI constraint F\*x <= B '''  
  
 def \_\_init\_\_(self, F, B):  
 self.F = F  
 self.F0 = B  
 self.Q = chol\_ext(len(self.F0))  
  
 def \_\_call\_\_(self, x):  
 n = len(x)  
  
\* def getA(i, j):  
\* return self.F0[i, j] - sum(  
\* self.F[k][i, j] \* x[k] for k in range(n))  
\*  
\* self.Q.factor(getA)  
 if self.Q.is\_spd():  
 return None  
 v, ep = self.Q.witness()  
 g = np.array([self.Q.sym\_quad(v, self.F[i])  
 for i in range(n)])  
 return g, ep

]

## Google Benchmark Comparison

2: ----------------------------------------------------------  
2: Benchmark Time CPU Iterations  
2: ----------------------------------------------------------  
2: BM\_LMI\_Lazy 131235 ns 131245 ns 4447  
2: BM\_LMI\_old 196694 ns 196708 ns 3548  
2/4 Test #2: Bench\_BM\_lmi ..................... Passed 2.57 sec

## Example - Matrix Norm Minimization

* Let
* Problem can be reformulated as
* Binary search on can be used for this problem.

.small[

class qmi\_oracle:  
 t = None  
 count = 0  
  
 def \_\_init\_\_(self, F, F0):  
 self.F = F  
 self.F0 = F0  
 self.Fx = np.zeros(F0.shape)  
 self.Q = chol\_ext(len(F0))  
  
 def update(self, t): self.t = t  
  
 def \_\_call\_\_(self, x):  
 self.count = 0; nx = len(x)  
  
 def getA(i, j):  
 if self.count < i + 1:  
 self.count = i + 1  
 self.Fx[i] = self.F0[i]  
 self.Fx[i] -= sum(self.F[k][i] \* x[k]  
 for k in range(nx))  
 a = -self.Fx[i].dot(self.Fx[j])  
 if i == j: a += self.t  
 return a  
  
 self.Q.factor(getA)  
 if self.Q.is\_spd(): return None  
 v, ep = self.Q.witness()  
 p = len(v)  
 Av = v.dot(self.Fx[:p])  
 g = np.array([-2\*v.dot(self.F[k][:p]).dot(Av)  
 for k in range(nx)])  
 return (g, ep)

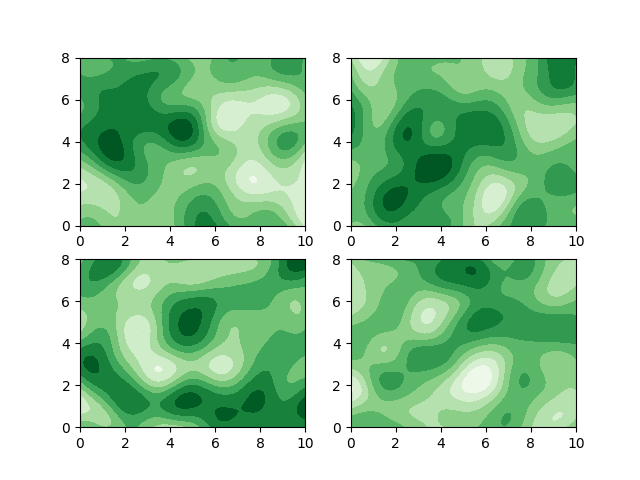
]

## Example - Estimation of Correlation Function

* Let , where
  + ’s are the unknown coefficients to be fitted
  + ’s are a family of basis functions.
* The covariance matrix can be recast as:
* where

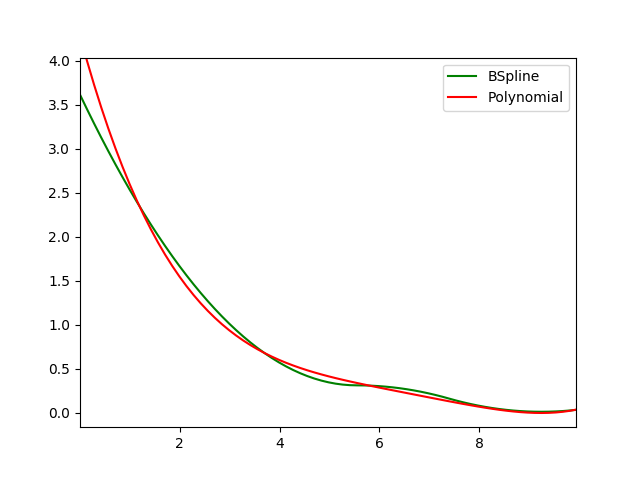
## 🔬 Experimental Result

.pull-left[



Data Sample (kern=0.5)

] .pull-right[

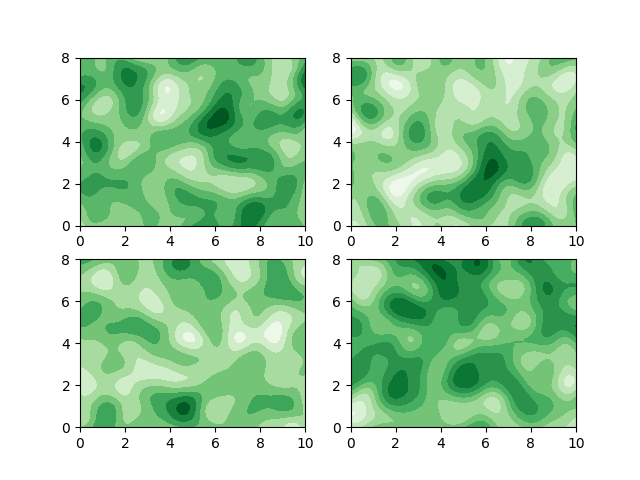


Least Square Result

]

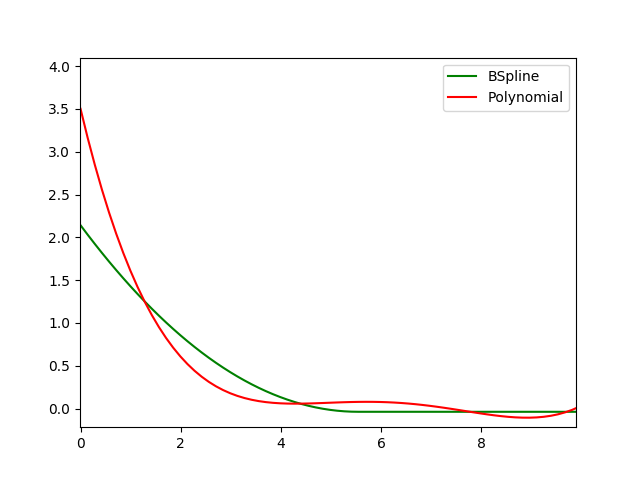
## 🔬 Experimental Result II

.pull-left[



Data Sample (kern=1.0)

] .pull-right[

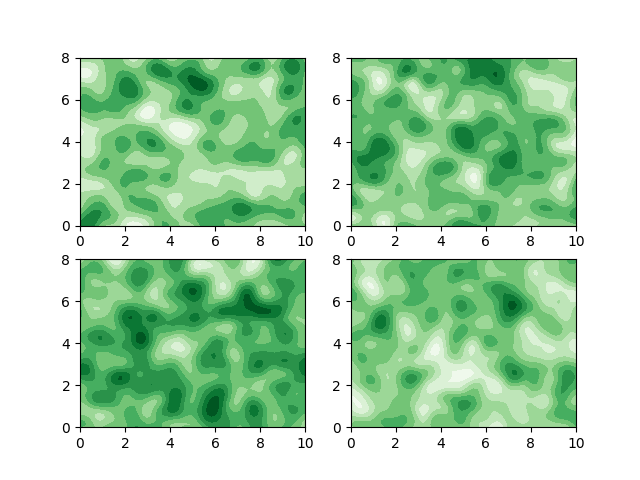


Least Square Result

]

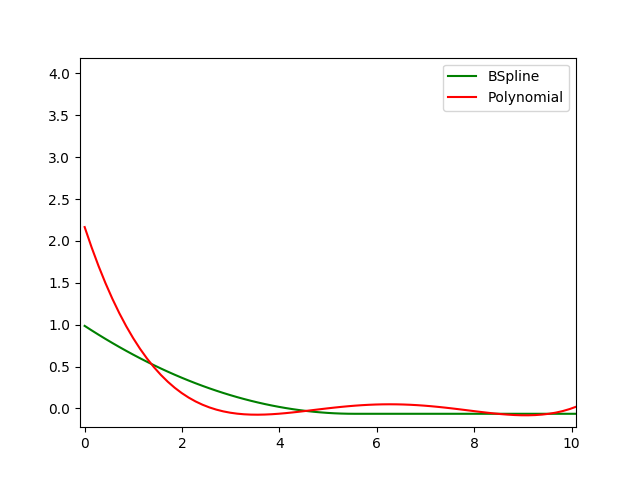
## 🔬 Experimental Result III

.pull-left[



Data Sample (kern=2.0)

] .pull-right[



Least Square Result

]

# Q & A 🗣️