Disciplined Geometric Programming: Introduction and Examples

Chee Wei TAN

Introduction

The Disciplined geometric programming (DGP) is a DCP software for log-log convex functions of positive variables:

https://www.cvxpy.org/tutorial/dgp

- "Disciplined geometric programming," Optimization Letters, A. Agrawal, S. Diamond and S. Boyd, 2019
- DGP is a ruleset for log-log convex programs (LLCPs), which are problems that are convex after the variables, objective functions, and constraint functions are replaced with their logs (log-log transformation)
- Every geometric program (GP)[1] and generalized geometric program (GGP) is an LLCP, but there are LLCPs that are neither GPs nor GGPs.

[1] Another freely-available software for solving GP: Stanford GGPLAB: https://web.stanford.edu/~boyd/ggplab

Introduction

CVXPY lets you form and solve DGP problems, just as it does for DCP problems. For example, the following code solves a simple geometric program,

```python
import cvxpy as cp

# DGP requires Variables to be declared positive via `pos=True`
x = cp.Variable(pos=True)
y = cp.Variable(pos=True)
z = cp.Variable(pos=True)

objective_fn = x * y * z
constraints = [
  4 * x * y * z + 2 * x * z <= 10, x <= 2*y, y <= 2*x, z >= 1]
problem = cp.Problem(cp.Maximize(objective_fn), constraints)
problem.solve(gp=True)
print("Optimal value: ", problem.value)
print("x: ", x.value)
print("y: ", y.value)
print("z: ", z.value)
```

Log-log Curvature

CVXPY's log-log curvature analysis can flag Expressions as unknown even when they are log-log convex or log-log concave. Note that any log-log constant expression is also log-log affine, and any log-log affine expression is log-log convex and log-log concave.

Log-log Curvature Analysis

The log-log curvature of an Expression is stored in its `.log_log_curvature` attribute. For example, running the following script

```
import cvxpy as cp

x = cp.Variable(pos=True)
y = cp.Variable(pos=True)

constant = cp.Constant(2.0)
monomial = constant * x * y
posynomial = monomial + (x ** 1.5) * (y ** -1)
reciprocal = posynomial ** -1
unknown = reciprocal + posynomial

print(constant.log_log_curvature)
print(monomial.log_log_curvature)
print(posynomial.log_log_curvature)
print(reciprocal.log_log_curvature)
print(unknown.log_log_curvature)
```

Prints the following output:
LOG-LOG CONSTANT
LOG-LOG AFFINE
LOG-LOG CONVEX
LOG-LOG CONCAVE
UNKNOWN

Log-log Curvature Analysis

If an Expression satisfies the composition rule, we colloquially say that the Expression "is DGP." You can check whether an Expression is DGP by calling the method `is_dgp()`. For example, the assertions in the following code block will pass.

```
import cvxpy as cp

x = cp.Variable(pos=True)
y = cp.Variable(pos=True)

monomial = 2.0 * constant * x * y
posynomial = monomial + (x ** 1.5) * (y ** -1)

assert monomial.is_dgp()
assert posynomial.is_dgp()
```

Log-log Curvature Analysis

You can also check the log-log curvature of an Expression by calling the methods `is_log_log_constant()`, `is_log_log_affine()`, `is_log_log_convex()`, `is_log_log_concave()`. For example, `posynomial.is_log_log_convex()` would evaluate to True.

```
import cvxpy as cp

x = cp.Variable(pos=True)
y = cp.Variable(pos=True)

monomial = 2.0 * constant * x * y
posynomial = monomial + (x ** 1.5) * (y ** -1)

assert monomial.is_dgp()
assert posynomial.is_dgp()
```

DGP Atoms

This section of the tutorial describes the DGP atom library, that is, the atomic functions with known log-log curvature and monotonicity. CVXPY uses the function information in this section and the DGP rules to mark expressions with a log-log curvature. Note that every DGP expression is positive.

- Infix operators: The infix operators $+, *, /$ are treated as atoms. The operators $*$ and $/$ are log-log affine functions. The operator $+$ is log-log convex in both its arguments.
- Transpose: Transpose is a log-log affine function.
- Power: Taking powers is a log-log affine function.

DGP Atoms

- Scalar functions: A scalar function takes one or more scalars, vectors, or matrices as arguments and returns a scalar.

Function	Meaning	Domain
geo_mean(x) geo_mean(x,p) $p \in \mathbf{R}^n_+$ $p \neq 0$	$x_1^{1/n} \cdots x_n^{1/n}$ $(x_1^{p_1} \cdots x_n^{p_n})^{\frac{1}{1^T p}}$	$x \in \mathbf{R}^n_+$
harmonic_mean(x)	$\frac{n}{\frac{1}{x_1}+\cdots+\frac{1}{x_n}}$	$x \in \mathbf{R}^n_+$
max(X)	$\max_{ij}\{X_{ij}\}$	$X \in \mathbf{R}^{m\times n}_{++}$
min(X)	$\min_{ij}\{X_{ij}\}$	$X \in \mathbf{R}^{m\times n}_{++}$
norm(x) norm(x,2)	$\sqrt{\sum_i \lvert x_i\rvert^2}$	$X \in \mathbf{R}^n_{++}$
norm(X,"fro")	$\sqrt{\sum_{ij} X_{ij}^2}$	$X \in \mathbf{R}^{m\times n}_{++}$
norm(X,1)	$\sum_{ij} \lvert X_{ij}\rvert$	$X \in \mathbf{R}^{m\times n}_{++}$

DGP Atoms

Function	Meaning	Domain
norm(X,"inf")	$\max_{ij}\{\lvert X_{ij}\rvert\}$	$X \in \mathbf{R}^{m\times n}_{++}$
pnorm(X,p) $p \geq 1$ or p='inf'	$\lVert X\rVert_p = (\sum_{ij} \lvert X_{ij}\rvert^p)^{1/p}$	$X \in \mathbf{R}^{m\times n}_{++}$
pnorm(X,p) $0 < p < 1$	$\lVert X\rVert_p = (\sum_{ij} X_{ij}^p)^{1/p}$	$X \in \mathbf{R}^{m\times n}_{++}$
prod(X)	$\prod_{ij} X_{ij}$	$X \in \mathbf{R}^{m\times n}_{++}$
quad_form(x,P)	$x^T P x$	$x \in \mathbf{R}^n$, $P \in \mathbf{R}^{n\times n}_{++}$
quad_over_lin(X,y)	$(\sum_{ij} X_{ij}^2)/y$	$x \in \mathbf{R}^n_{++}$, $y > 0$
sum(X)	$\sum_{ij} X_{ij}$	$X \in \mathbf{R}^{m\times n}_{++}$
sum_squares(X)	$\sum_{ij} X_{ij}^2$	$X \in \mathbf{R}^{m\times n}_{++}$
trace(X)	tr(X)	$X \in \mathbf{R}^{n\times n}_{++}$
pf_eigenvalue(X)	spectral radius of X	$X \in \mathbf{R}^{n\times n}_{++}$

DGP Atoms

- Elementwise functions:

Function	Meaning	Domain
diff_pos(x,y)	$x - y$	$0 < y < x$
entr(x)	$-x\log(x)$	$0 < x < 1$
exp(x)	e^x	$x > 0$
log(x)	$\log(x)$	$x > 1$
maximum(x,y)	$\max\{x, y\}$	$x, y > 0$
minimum(x,y)	$\min\{x, y\}$	$x, y > 0$
multiply(x,y)	$x * y$	$x, y > 0$
one_minus_pos(x)	$1 - x$	$0 < x < 1$
power(x,0)	1	$x > 0$
power(x,p)	x	$x > 0$
sqrt(x)	$\sqrt{x}$	$x > 0$
square(x)	x^2	$x > 0$

DGP Atoms

- Vector/matrix functions:

Function	Meaning	Domain
bmat([[X11, ...,X1q], ...[Xp1, ...,Xpq]])	$\begin{bmatrix} X^{(1,1)} & \dots & X^{(1,q)} \\ \vdots & & \vdots \\ X^{(p,1)} & \dots & X^{(p,q)} \end{bmatrix}$	$X^{(i,j)} \in \mathbf{R}_{++}^{m_i \times n_j}$
diag(x)	$\begin{bmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{bmatrix}$	$x \in \mathbf{R}_{++}^{n}$
diag(X)	$\begin{bmatrix} X_{11} \\ \vdots \\ X_{nn} \end{bmatrix}$	$x \in \mathbf{R}_{++}^{n \times n}$
eye_minus_inv(X)	$(I - X)^{-1}$	$X \in \mathbf{R}_{++}^{n \times n}$, $\lambda_{pf}(X) < 1$

DGP Atoms

Function	Meaning	Domain
hstack([X$_1$, ..., X_k])	$[X^{(1)} \cdots X^{(k)}]$	$X^{(i)} \in \mathbf{R}^{m \times n_i}_{++}$
matmul(X,Y)	XY	$X \in \mathbf{R}^{m \times n}_{++}$, $Y \in \mathbf{R}^{n \times p}_{++}$,
resolvent(X)	$(sI - X)^{-1}$	$X \in \mathbf{R}^{n \times n}_{++}$, $\lambda_{pf}(X) < s$
reshape(X, (n',m'))	$X' \in \mathbf{R}^{m' \times n'}$	$X \in \mathbf{R}^{m \times n}_{++}$ $m'n' = mn$
vec(X)	$x' \in \mathbf{R}^{mn}$	$X \in \mathbf{R}^{m \times n}_{++}$
vstack([X$_1$, ..., X_k])	$\begin{bmatrix} X^{(1)} \\ \vdots \\ X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbf{R}^{m_i \times n}_{++}$

Example 1: Maximizing the Volume of a Box

In this example, we maximize the shape of a box with height h, width w, and depth w, with limits on the wall area $2(hw + hd)$ and the floor area wd, subject to bounds on the aspect ratios h/w and w/d. The optimization problem is

$$
\begin{array}{rl}
\text{maximize} & hwd \\
\text{subject to} & 2(hw + hd) \leq A_{wall} \\
& wd \leq A_{flr} \\
& \alpha \leq h/w \leq \beta \\
& \gamma \leq d/w \leq \delta
\end{array}
$$

Example 1: Maximizing the Volume of a Box

```python
import cvxpy as cp

# Problem data.
A_wall, A_flr= 100, 10
alpha, beta, gamma, delta= 0.5, 2, 0.5, 2

h = cp.Variable(pos=True, name="h")
w = cp.Variable(pos=True, name="w")
d = cp.Variable(pos=True, name="d")

volume = h * w * d
wall_area = 2 * (h * w + h * d)
flr_area = w * d
hw_ratio = h/w
dw_ratio = d/w
constraints = [
    wall_area <= A_wall,
    flr_area <= A_flr,
    hw_ratio >= alpha,
    hw_ratio <= beta,
    dw_ratio >= gamma,
    dw_ratio <= delta
]
problem = cp.Problem(cp.Maximize(volume), constraints)
assert not problem.is_dcp()
assert problem.is_dgp()
problem.solve(gp=True)
```

Example 2: Perron-Frobenius Matrix Completion

In this problem, we are given some entries of an elementwise positive matrix A, and the goal is to choose the missing entries so as to minimize the Perron-Frobenius eigenvalue or spectral radius. Letting Ω denote the set of indices (i, j) for which A_{ij} is known, the optimization problem is

$$\begin{array}{ll} \text{minimize} & \lambda_{pf}(X) \\ \text{subject to} & \prod_{(i,j) \notin \Omega} X_{ij} = 1 \\ & X_{ij} = A_{ij}, (i, j) \in \Omega \end{array}$$

which is a log-log convex program. Below is an implementation of this problem, with specific problem data

$$A = \begin{bmatrix} 1.0 & ? & 1.9 \\ ? & 0.8 & ? \\ 3.2 & 5.9 & ? \end{bmatrix},$$

Example 2: Perron-Frobenius Matrix Completion

```
import cvxpy as cp

n = 3
known_value_indices = tuple(zip(*[[0, 0], [0, 2], [1, 1], [2, 0], [2, 1]]))
known_values = [1.0, 1.9, 0.8, 3.2, 5.9]
X = cp.Variable((n, n), pos=True)
objective_fn = cp.pf_eigenvalue(X)
constraints = [
  X[known_value_indices] == known_values,
  X[0, 1] * X[1, 0] * X[1, 2] * X[2, 2] == 1.0,
]
problem = cp.Problem(cp.Minimize(objective_fn), constraints)
problem.solve(gp=True)
print("Optimal value: ", problem.value)
print("X:\n", X.value)
```

```
Optimal value:  4.702374203221372
X:
 [[1.         4.63616907 1.9       ]
 [0.49991744 0.8        0.37774148]
 [3.2        5.9        1.14221476]]
```

Example 3: Power Control

This example formulates and solves a power control problem for communication systems, in which the goal is to minimize the total transmitter power across n transmitters, each trasmitting positive power levels $P_1, P_2, \cdots, P_n$ to n receivers, labeled $1, \cdots, n$, with receiver i receiving signal from transmitter i.

The power received from transmitter j at receiver i is $G_{ij}P_j$, where $G_{ij} > 0$ represents the path gain from transmitter j to receiver i.

The signal power at receiver i is $G_{ii}P_i$, and the interference power at receiver i is $\sum_{k\neq i} G_{ik}P_k$. The noise power at receiver i is σ_i, and the signal to noise ratio (SINR) of the ith receiver-transmitter pair is

$$S_i = \frac{G_{ii}P_i}{\sigma_i + \sum_{k\neq i} G_{ik}P_k}.$$

Example 3: Power Control

The transmitters and receivers are constrained to have a minimum SINR S^{min}, and the P_i are bounded between P_i^{min} and P_i^{max}. This gives the problem

$$\begin{array}{rl} \text{minimize} & P_1 + \cdots + P_n \\ \text{subject to} & P_i^{min} \leq P_i \leq P_i^{max}, \\ & 1/S^{min} \geq \dfrac{\sigma_i + \sum_{k \neq i} G_{ik} P_k}{G_{ii} P_i} \end{array}$$

.

Example 3: Power Control

```
import cvxpy as cp
import numpy as np

# Problem data
n = 5                        # number of transmitters and receivers
sigma = 0.5 * np.ones(n)     # noise power at the receiver i
p_min = 0.1 * np.ones(n)     # minimum power at the transmitter i
p_max = 5 * np.ones(n)       # maximum power at the transmitter i
sinr_min = 0.1               # threshold SINR for each receiver

# Path gain matrix
G = np.array(
   [[1.0, 0.1, 0.2, 0.1, 0.05],
    [0.1, 1.0, 0.1, 0.1, 0.05],
    [0.2, 0.1, 1.0, 0.2, 0.2],
    [0.1, 0.1, 0.2, 1.0, 0.1],
    [0.05, 0.05, 0.2, 0.1, 1.0]])
p = cp.Variable(shape=(n,), pos=True)
objective = cp.Minimize(cp.sum(p))

S_p = []
for i in range(n):
    S_p.append(cp.sum(cp.hstack(G[i, k]*p for k in range(n) if i != k)))
S = sigma + cp.hstack(S_p)
signal_power = cp.multiply(cp.diag(G), p)
inverse_sinr = S/signal_power
constraints = [
    p >= p_min,
    p <= p_max,
    inverse_sinr <= (1/sinr_min),
]

problem = cp.Problem(objective, constraints)
problem.solve(gp=True)
problem.value
```

Example 4: Rank-one Nonnegative Matrix Factorization

We would like to approximate A as the outer product of two positive vectors x and y, with x normalized so that the product of its entries equals 1. Our criterion is the average relative deviation between the entries of A and xy^T, that is,

$$\frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n} R(A_{ij}, x_i y_j),$$

where R is the relative deviation of two positive numbers, defined as

$$R(a, b) = \max\{a/b, b/a\} - 1$$

Example 4: Rank-one Nonnegative Matrix Factorization

The corresponding optimization problem is

$$\begin{array}{ll} \text{minimize} & \dfrac{1}{mn}\displaystyle\sum_{i=1}^{m}\sum_{j=1}^{n} R(X_{ij}, x_i y_j) \\ \text{subject to} & x_1 x_2 \cdots x_m = 1 \\ & X_{ij} = A_{ij}, \quad \text{for } (i,j) \in \Omega, \end{array}$$

with variables $X \in \mathbb{R}^{mn}_{++}$, $x \in \mathbb{R}^{m}_{++}$, and $y \in \mathbb{R}^{n}_{++}$. We can cast this problem as an equivalent generalized geometric program by discarding the 1 from the relative deviations.

The below code constructs and solves this optimization problem, with specific problem data

$$A = \begin{bmatrix} 1.0 & ? & 1.9 \\ ? & 0.8 & ? \\ 3.2 & 5.9 & ? \end{bmatrix},$$

Example 4: Rank-one Nonnegative Matrix Factorization

```python
import cvxpy as cp

m = 3
n = 3
X = cp.Variable((m, n), pos=True)
x = cp.Variable((m,), pos=True)
y = cp.Variable((n,), pos=True)

outer_product = cp.vstack([x[i] * y for i in range(m)])
relative_deviations = cp.maximum(
  cp.multiply(X, outer_product ** -1),
  cp.multiply(X ** -1, outer_product))
objective = cp.sum(relative_deviations)
constraints = [
  X[0, 0] == 1.0,
  X[0, 2] == 1.9,
  X[1, 1] == 0.8,
  X[2, 0] == 3.2,
  X[2, 1] == 5.9,
  x[0] * x[1] * x[2] == 1.0,
]
problem = cp.Problem(cp.Minimize(objective), constraints)
problem.solve(gp=True)
```

Reference

https://www.cvxpy.org/index.html

https://web.stanford.edu/~boyd/papers/pdf/gp_tutorial.pdf

https://www.cvxpy.org/tutorial/dgp/index.html