Google Summer of Code 2023 : NumFOCUS Final Report on Adding structured Lagrangian support to CVXPY

Aryaman Jeendgar

August 28, 2023

Contents

1 Introduction:

This document details the contributions I made to *CVXPY's* codebase over the duration of my GSoC, '23 project titled: **Adding Structured Lagrangian support to CVXPY**. My work during the GSoC period was to build infrastructure for verifying the KKT conditions, in-so-far as that is concerned, we were able to implement it for all differentiable `Atom` classes that CVXPY supports (for the case of *stationarity checks*).

I provide the KKT conditions (and their corresponding CVXPY method) below. Again, consider a general optimization problem **P**, and it's dual **D**:

$$
\begin{array}{lll}
 & \textbf{P} & \\
\max\limits_{x \in \mathcal{D}} & f_0(x) & \\
\text{subject to} & f_i(x) \leq 0, & i = 1 \ldots, m \\
 & h_i(x) = 0, & i = 1, \ldots, p
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{D} \\
\max\limits_{\lambda, \nu} & g(\lambda, \nu) \\
\text{subject to} & \lambda \geq 0
\end{array}
$$

Where, $g(\lambda, \nu) = \min_x \left(f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x)\right)$

1. **Stationarity of the Lagrangian**,
   `cvxpy/tests/solver_test_helpers.py:SolverTestHelper::`
   `check_stationary_lagrangian`

   $$\mathbf{0} \in \partial \left(f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x) \right) \tag{1}$$

   Here $\partial(.)$ represents the subdifferential of a function.

2. **Dual feasibility**,
   `cvxpy/tests/solver_test_helpers.py:SolverTestHelper::`
   `check_dual_domains`:

   $$\lambda_i \geq 0, i = 1, 2, \ldots, m \tag{2}$$

3. **Complimentary Slackness**,
`cvxpy/tests/solver_test_helpers.py:SolverTestHelper::check_complimentarity` (existed pre-GSoC contributions):

$$\lambda_i . f_i(x) = 0, i = 1, 2, \dots, m \tag{3}$$

4. **Primal feasibility**,
`cvxpy/tests/solver_test_helpers.py:SolverTestHelper::check_primal_feasibility` (existed pre-GSoC contributions):

$$f_i(x) \leq 0, i = 1, 2, \dots, m$$
$$h_i(x) = 0, i = 1, 2, \dots, p$$

The implementation of all of the above constituent conditions, with the exception of `check_stationary_lagrangian` and `check_dual_domains` (for certain `Constraint` classes) is *complete* and general enough to encompass the majority of CVXPY's current functionality.

A lot of time was spent thinking about the particular design and implementation of the above methods since it involved major changes to the public API of CVXPY. Most of the newly implemented methods and changes will possibly be made public after the above KKT verification methods are made sufficiently general to cover all of CVXPY's current functionality.

The plan is to move the above verification methods out of `SolverTest-Helpers` (perhaps temporarily to `cvxpy/problems/kkt.py`), and eventually write an instance method on the `cp.Problem` class that checks all of the KKT conditions and provides meaningful output in case any of the checks fail.

2 Coding Period:

2.1 Pre-midterm evaluation:

The major contribution pre-midterm was the implementation of the `check_stationary_lagrangian` method.

Much of this has been documented in a blog post that can be found on my webpage, **here**.

But to summarize, we were able to quickly get up and running with a basic implementation that worked for the constraints in CVXPY that support recovery of the values of the dual variables (this is done via `cp.Constraint-` `-.dual_variables` to access the dual variables themselves or via `cp.Constraint-` `.dual_value` to access the values that they store. These fields are populated after a `cp.Problem` is succesfully solved).

The tricky part with this implementation was the fact that CVXPY supports passing in implicit constraints via flags when declaring variables. Since there was no way to symbolically access these constraints, we had to be a bit clever and re-contextualize stationarity in a more general way to account for these implicit constraints. Namely the idea being that of characterizing the dual cone K^* of the convex cone K that the variable is implicitly constrained to lie in. Specifically, instead of the gradient w.r.t these variables being $\mathbf{0}$, the gradient should instead lie within K^* — hence, we had to check set membership for the computed gradients for variables with such flags.

CVXPY supports the following flags:

```
nonneg : bool
nonpos : bool
symmetric : bool
diag : bool
PSD : bool
NSD : bool
pos : bool
neg : bool
```

Of these, the `diag` attribute (the dual to the set of all diagonal matrices is the set of all *hollow* matrices) is currently un-supported, because of their peculiar internal representation within CVXPY (via SciPy `CSC` matrices).

Another important point of note while implementing the stationarity check was in the construction of the lagrangian itself, particularly in inferring the correct sign convention for different kinds of CVXPY constraints. Of note as a consequence of playing around with possible combinations came the deprication of the `NonPos` constraint (#2155), which followed from the discussion, here. The reason for this was the inconsistent sign of the dual variables that CVXPY was computing for `NonPos` constraints, i.e. the dual variables recovered were always non-negative, whereas they ought to be non-positive (since the `NonPos` cone is self-dual), this was due to a long-standing convention of ensuring that `(expr < = 0).dual_value` matches `NonPos(expr).dual_value`, and hence, while the correct sign convention for moving contributions from conic constraints into the lagrangian was `-(contrib)`, for `NonPos` constraints it was `+(contrib)`.

I also made a minor bug-fix which I happened upon when I was working on the implementation of dual cones for the existing `Constraint` classes. **NOTE**: The test cases for much of this functionality resides within `cvxpy-/tests/test_kkt.py`

2.2 Post-midterm evaluation:

There were two major threads that I worked on post-midterm:

2.2.1 #2204, `check_dual_domains`, and the introduction of the `dual_cone/residual` methods:

This PR introduces several sweeping API altercations. For one, it introduces an entirely *new* `Cone` class which subclasses `Constraint`. This is a new class which all convex conic constraints now inherit from.

The reason for this change was the introduction of two new methods (which do not make any semantic sense for algebraic constraints, such as `Inequality`), namely, `_dual_cone` (private method) and `dual_residual`. The former returns the corresponding dual cone of the `Cone` instance, while the latter is just a convenient method for computing the violation on the dual variable values that CVXPY returns w.r.t K^* returned in `_dual_cone` — we use `dual_residual` within `check_dual_domains`.

2.2.2 WiP on fork, `_is_differentiable_at` — verifying points of non-differentiability:

This PR again adds a new method, this time, to the `Atom` class. The intention behind this work was the introduction of the notion of `strict_-differentiability` for `grad` computations in CVXPY.

Namely, CVXPY returns *a* subgradient for every atom (for which `grad` has been implemented), but for points of non-differentiability, the subdifferential at that point is a non-singleton set, and hence, in such cases a distinction needed to be made to the end of stationarity checks.

This PR adds several new pieces of code, for one, we implement the `_is_differentiable_at` method on a variety of atoms, which point out whether an atom is differentiable or not at that point (i.e., whether or not the subdifferential at that point is a singleton set). To keep the existing `grad` computation as parallel to the current semantics as possible, we introduce a new context manager within `cvxpy/utilities/scopes.py::-strict_differentiability_scope`. Here is an example use case:

```
import cvxpy as cp
import numpy as np
from cvxpy.utilities.scopes import strict_differentiability_scope

X = cp.Variable(shape=(3,3))
X.value = np.zeros((3,3))
expr = cp.norm1(X)

expr.grad # does not throw an error
with strict_differentiability_scope():
    expr.grad # throws NotDifferentiableError
```

The reason the `_is_differentiable_at` method was wrapped around the `_grad` methods on every atom (and not directly on the `grad` method that is defined on the `Atom` class which implements the chain rule), is to ensure that we can naturally leverage the recursive *canonicalization* process of CVXPY parsing the expression tree (since `_grad` is where the computation for `grad` bottoms out). Similarly, in the case of `AxisAtom`'s, we wrap the `_is_differentiable_at` method around the `_column_grad` method.

This is still a WiP, with some the `_is_differentiable_at` method for some atom classes requiring some changes.

2.3 Future work:

After the official period for GSoC 2023 ends, I will be extending our work as part of my thesis requirement with Dr. Riley Murray. So far, we plan on working on the following major features:

1. Introduce a new `ConvexSet` class to the end of implementing subdifferential support in CVXPY from the ground up

2. Implement dual variable recovery, and the `_dual_cone` and `dual_residual` methods for `PowConeND`

3. Derive the dual cone for the semidefinite approximation of the class of exotic convex cones (namely, the Relative Entropy Cone and the Operator Relative Entropy Cone) which we worked on implementing within CVXPY as part of my GSoC 2022 project.