Design Document

Code Generation in CVXPY

1 Introduction

Context

CVXPY allows to formulate convex optimization problems in a natural (mathematical) way and to solve them. By default, QPs, SOCPs, and cone programs are solved with the OSQP, ECOS, and SCS method, respectively. Generating compilable C code helps to solve these problems even faster. There exist code generation utilities for the individual methods. However, the whole process from user data, to data in canonical form, to the canonical solution (code generation exists), back to the user solution is not yet readily generated in C.

Goals

The main goal of this project is to have code generation functionality in CVXPY for the whole affine-solve-affine (ASA) procedure. In particular, code generation functionality for the affine mappings needs to be implemented. Consequently, users should be able to use the code in their python notebook, and in a separate C project.

Document Structure

In a top-down way, based on the user interface described in Section 2, the functional design is detailed in Section 3 and Section 4 derives the technical design. Software development milestones are set in Section 5. Section 6 gives an outlook to additional functionality and follow-up research possibilities. W.l.o.g., this document focuses on QPs and the OSQP method. In this document, the word ‘solver‘ denotes a single solver instance associated with a certain method and problem family.

2 User Interface

The targeted user interface is best explained with an example, with the new elements highlighted in green.

```
import cvxpy as cp

m, n = 3, 2
x = cp.Variable((n, 1))
F = cp.Parameter((m, n))
g = cp.Parameter((m, 1))
gamma = cp.Parameter(nonneg=True)

cost = cp.sum_squares(F @ x - g) + gamma * cp.sum_squares(x)
constraints = [x >= 0]
myProblem = cp.Problem(cp.Minimize(cost), constraints)

myProblem.solve(codegen=True, codedir="myDirectory", ...)
print(x.value)
```

If the user wishes to solve a problem via code generation, they should just have to set the `codegen` option to **`True`** in addition to the known way of setting up and solving problems in CVXPY. Even the `codedir` argument (specifying where to store the generated code) would default to a name like **"`CVXPY_codegen`"**. To allow for more engineering, the `solve(codegen=True)` function should decompose into the following cascade, of which each function should be usable separately:

```
myProblem.solve(codegen=True, codedir="myDirectory")
    myProblem.compileCode(codedir=codedir, target="pymodule")
        myProblem.generateCode(codedir=codedir)
```

3 Functional Design

Let us follow the above cascade from bottom to top.

`generateCode()` produces C code in the project folder **"`myDirectory`"**. The input to the main C function should be the problem data in user form (generally non-canonical). The output should be the solution in user form. After calling this function, the user should be able to add the generated files to some overall C project that involves solving an optimization problem.

`compileCode()` compiles the C project inside **"`myDirectory`"**. The `target` argument specifies whether to have the binary ASA-solver as a python module for the CVXPY `solve()` function (**`target="pymodule"`**), similar to how it is done in OSQP code generation, or to create a standalone executable, for example.

`solve(codegen=True)` solves the optimization problem using the compiled python module.

4 Technical Design

5 Development Milestones

6 Outlook