QCML version 0.1
A Python Toolbox for Matrix Stuffing

Eric Chu
echu508@stanford.edu

Stephen Boyd
boyd@stanford.edu

October 28, 2013

QCML is a Python module that provides a simple and lightweight domain specific language (DSL) for describing convex optimization problem families representable as second-order cone problems (SOCP). It is a code generator that produces lightweight Python, Matlab, or C code for *matrix stuffing*. This allows one to extend SOCP solvers in these langauges to a variety of problem classes. It is intended to complement the paper, *Code Generation for Embedded Second-order Cone Programming* [1], although the main ideas in its implementation are well-documented in *Graph Implementation for Nonsmooth Convex Programs* [2].

QCML comes with a small example library and consists of a single Python object, QCML, with four methods

- parse, which parses an optimization problem specified as a string and checks that it is convex,
- canonicalize, which symbolically converts the problem into a second-order cone program,
- codegen, which produces source code that performs the transformation from problem-specific data to second-order cone problem data and back, and
- save, which saves the resulting source code into a folder for future use.

The codegen method can be used to generate Python, C, and Matlab source code.

Some important features:

- QCML is similar in spirit to CVX, CVXGEN, and CVXPY. Unlike these packages, however, QCML exposes and generates code for matrix stuffing, allowing the resulting code to be paired with any solver that targets a standard SOCP.
- QCML is capable of generating fully abstract code with problem data and dimensions as input to the *generated* code, although it can specialize the code if given problem dimensions before code is generated.
- QCML supports matrix parameters with arbitrary sparsity patterns.

That said, there are several important caveats:

- QCML is intended for use by *developers* wishing to employ convex optimization in their source code with little to no overhead from the modeling layer. Typically, these are users familiar with CVX or CVXPY who wish to do away with modeling overhead in their applications.

- `QCML` is not a full-featured modeling language. Several features are conspicuously missing and unless specifically requested, will remain missing.
- For general purpose convex optimization in Python with more flexibility than `QCML`, we recommend `CVXPY` (`http://github.com/cvxgrp/cvxpy`).
- For general purpose convex optimization in Matlab, we recommend `CVX` (`http://cvxr.com`).

Please send your feedback or report any bugs through our Github repository.

The rest of this document covers the basic use of `QCML` and walks through a full example with a lasso (ℓ_1-regularized least-squares) problem. `QCML` is licensed under BSD.

Contents

1 Installing QCML

QCML depends on the following:

- Python 2.7.2+ (no Python 3 support)
- PLY, the Python Lex-Yacc parsing framework, available as the `python-ply` or `py-ply` package in most distributions.
- ECOS (`http://github.com/ifa-ethz/ecos`)
- NUMPY (`http://numpy.org`)
- SCIPY (`http://scipy.org`)

For (some) unit testing, we use Nose (`http://nose.readthedocs.org`).

Once these dependencies are installed, typically through a package manager, QCML can be installed by typing

```
$ python setup.py install
```

at the command line. The unit tests can be run with `nosetests`.

2 Basic usage

A QCML problem must be specified following the language described in §3, but we give a basic example of QCML here. The QCML API consists of four main functions, `parse`, `canonicalize`, `codegen`, and `save`. All optimization problems in QCML are specified as strings; suppose `prob` is a string that represents a particular optimization problem, then basic usage is summarized in the following Python code:

```
from qcml import QCML
p = QCML()              # creates a QCML object
p.parse(prob)           # verifies that ‘prob’ is a valid QCML string
p.canonicalize()        # converts the problem into standard form
p.codegen("python")     # generates Python source code for transforming ‘prob’
p.save("myprob")        # saves the source code into a folder called ‘myprob’
```

The `parse` step verifies that the optimization problem adheres to the rules described in §3.

The `canonicalize` step converts a QCML problem into the standard conic form

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Gx + s = h \\ & Ax = b \\ & s \in \mathcal{K}, \end{array} \tag{1}$$

where $x \in \mathbf{R}^n$ and $s \in \mathbf{R}^m$ are the optimization variables, $c \in \mathbf{R}^n$, $h \in \mathbf{R}^m$, $b \in \mathbf{R}^p$, $G \in \mathbf{R}^{m\times n}$, $A \in \mathbf{R}^{p\times n}$, and the convex cone $\mathcal{K}$ are problem data. The generated code is thus compatible with any solver that solves (1). We do not give more detail about canonicalization in this user guide; for more detail, consult [3, 2, 1].

The `codegen` and `save` steps target three different languages, Python, C, and Matlab. We detail these in §6.

3 QCML language

Optimization problems in `QCML` are specified as *strings* which must adhere to the rules of the `QCML` modeling language. The `parse` function checks that the problem string adheres to these rules. We describe that language and its rules here.

3.1 Types

There are three basic types in `QCML`:

- `dimension` (or `dimensions` for multiple dimension declarations)
- `parameter` (or `parameters` for multiple parameter declrations)
- `variable` (or `variables` for multiple variable declarations)

Objects named and declared in this fashion are entirely *abstract.* Numeric literals such as `2`, `0.52`, *etc.*, in `QCML` are *concrete* and can be used directly in the `QCML` problem string.

The `dimension` objects are used to specify the (optional) sizes of `variable`s and `parameter`s. So a single `variable` or `parameter` object can represent an collection (or array) of scalar variables or parameters with abstract dimension. (For more detail on abstract dimensions, consult §5.) At the moment, `variables` can represent vectors (one-dimensional arrays), and `parameters` can represent vectors or matrices (two-dimensional arrays). If a `parameter` represents a matrix, it is assumed to represent a *sparse* matrix.

A `parameter` may optionally take a sign modifier (`positive` or `negative`). These are shorthand for 'nonnegative' and 'nonpositve', respectively.

To declare an object, give its type, followed by its name, and (optionally) its size: for instance,

```
variable x(n)
```

declares a vector variable named `x` with length `n`. Integer literals can also be used in place of dimensions. A more involved example is the following code

```
dimensions m n
variables x(n) y(m) z(5)
parameter A(m,n) positive
parameters b c(n)
```

which declares two dimensions, `m` and `n`; three variables `x`, `y`, and `z` of length `n`, `m`, and `5`, respectively; an elementwise positive (sparse) parameter matrix, `A`; the scalar parameter `b`; and the vector parameter `c` with dimension `n`.

3.2 Operators

`QCML` provides a set of operators for use with modeling. These are used in conjunction with the basic `variable` and `parameter` types to construct (affine) expressions. The standard (infix) operators are: `+`, `-`, `*`, and `/`. We also provide the (postfix) transpose operator: `'`, following Matlab convention. The `-` operator can also be used as a prefix to denote negation.

While the addition, subtraction, transposition, and negation operators behave as standard linear operators and can act on any objects, the multiplication and division operators are more restrictive:

- The division operator `/` has the restriction that both operands must be numeric constants.
- The multiplication operator `*` has the restriction that the *lefthand size* must be a parameter.

These restrictions are in place to ensure that expressions are *affine* (or linear) as a function of the variable objects (*i.e.*, variables).

3.3 Functions

`QCML` also provides a set of atoms (or functions) which take expressions constructed from the basic operations as arguments. These functions have additional properties associated with them:

- the *sign* of the output expression (either positive, negative, or unknown),
- the *monotonicity* of the function in each argument (either increasing, decreasing, or nonmonotone),
- and the *curvature* of the function (either convex, concave, or affine).

Roughly speaking, these can be associated with the sign of the zeroth, first, and second derivatives of the function, respectively. (Although not all functions we provide are differentiable.) More importantly, the sign and monotonicities of a function may change depending on the sign of the input. For instance, the `square` function has positive sign (its outputs are always positive) and is typically a nonmonotone function. However, when its input is restricted to be positive, it is an increasing function. When a *convex* function has this property, we say that its monotonicity is *signed*. This means it is increasing if the argument is positive, decreasing if the argument is negative, and nonmonotone otherwise. When a *concave* function has a signed monotonicity, it means that it is decreasing if the argument is positive, increasing if the argument is negative, and nonmonotone otherwise.

Before listing the atoms, a slight word on notation. We deviate slightly from standard notation and use $f : \mathbf{R}^p \to \mathbf{R}^q$ to mean that a function f takes as input a real p-vector and produces a real q-vector as output. Valid inputs are a *subset* of all possible p-vectors, specifically, its domain which we denote with $\mathbf{dom}\, f$. In other words, the notation $f : \mathbf{R}^p \to \mathbf{R}^q$ can be thought of in the computer science way—as a *type signature* for the function f. As an example, the log function has the following type signature, $\log : \mathbf{R} \to \mathbf{R}$, although $\mathbf{dom}\, f$ is the set of strictly positive reals.

3.3.1 Scalar atoms

A scalar atom is a function with the type signature $f : \mathbf{R} \to \mathbf{R}$. Table 1 lists out all the scalar atoms in our library and their signs, monotonicity, and curvatures.

The `pow_rat` atom is slightly special in that it takes two *integer literal* arguments, `p` and `q`, which must be in the set $\{1, 2, 3, 4\}$. The monotonicity and curvature of the atom depend on the values of `p` and `q` chosen. If $p > q$, the function is increasing and convex. If $p < q$, the function is decreasing and concave. If $p = q$, the function is affine and, moreover, is the identity function.

We can define the *extension*, $\tilde{f} : \mathbf{R}^n \to \mathbf{R}^n$, of a scalar atom f to a vector argument $x = (x_1, x_2, \ldots, x_n)$ as

$$\tilde{f}(x) = (f(x_1), f(x_2), \ldots, f(x_n)),$$

Function	Definition	Sign	Monotonicity	Curvature
`abs(x)`	$f(x) = \|x\|$	positive	signed	convex
`huber(x)`	$f(x) = \begin{cases} x^2 & \|x\| \leq 1 \\ 2\|x\| - 1 \leq 1 & \|x\| > 1 \end{cases}$	positive	signed	convex
`inv_pos(x)`	$f(x) = 1/x,\ x > 0$	positive	decreasing	convex
`neg(x)`	$f(x) = \max(-x, 0)$	positive	increasing	convex
`pos(x)`	$f(x) = \max(x, 0)$	positive	increasing	convex
`pow_rat(x,p,q)`	$f(x) = x^{p/q},\ x > 0$	positive	(depends)	(depends)
`square_over_lin(x,y)`	$f(x) = x^2/y,\ y > 0$	positive	signed, decreasing	convex
`square(x)`	$f(x) = x^2$	positive	signed	convex
`geo_mean(x,y)`	$f(x,y) = \sqrt{xy},\ x, y > 0$	positive	increasing, increasing	concave
`sqrt(x)`	$f(x) = \sqrt{x},\ x > 0$	positive	increasing	concave

Table 1: A list of the scalar atoms in `QCML`. Scalar atoms with multiple arguments require the same dimension for both arguments; their montonicities are listed separately.

i.e., $\tilde{f}$ is f applied elementwise. Whenever scalar functions are called with vector arguments, this extension is automatically applied. For scalar functions with the signature $f : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$, their extension has the signature $\tilde{f} : \mathbf{R}^n \times \mathbf{R}^n \to \mathbf{R}^n$.

3.3.2 Vector atom

A vector atom is a function with the type signature $f : \mathbf{R}^n \to \mathbf{R}$. Table 2 lists out all the scalar atoms in our library and their signs, monotonicity, and curvatures.

The `quad_over_lin` atom is slightly special in that it has a mixed type signature, $f : \mathbf{R}^n \times \mathbf{R} \to \mathbf{R}$. Its second argument is *always* scalar. If a nonscalar is provided, an exception will be raised.

The `max` and `min` atoms have signs which depend on the sign of their arguments. The rule for

Function	Definition	Sign	Monotonicity	Curvature
`quad_over_lin(x,y)`	$f(x) = x^T x/y,\ y > 0$	positive	signed, decreasing	convex
`norm_inf(x)`	$f(x) = \\|x\\|_\infty$	positive	signed	convex
`norm1(x)`	$f(x) = \\|x\\|_1$	positive	signed	convex
`norm(x)` or `norm2(x)`	$f(x) = \\|x\\|_2$	positive	signed	convex
`max(x)`	$f(x) = \max_i x_i$	(depends)	increasing	convex
`min(x)`	$f(x) = \min_i x_i$	(depends)	increasing	concave

Table 2: A list of the vector atoms in `QCML`. Vector atoms with multiple arguments require the same dimension for both arguments; their montonicities are listed separately.

the sign of `max` is as follows:

- if *any* of its arguments are positive `max` is positive,
- if *all* its arguments are negative `max` is negative,
- otherwise, `max` has unknown sign.

The rule for the sign of `min` is as follows:

- if *any* of its arguments are negative `min` is negative,
- if *all* its arguments are positive `min` is positive,
- otherwise, `min` has unknown sign.

With the exception of `quad_over_lin`, vector atoms can take variable arguments. The *extension*, $\tilde{f} : \mathbf{R}^n \times \mathbf{R}^n \to \mathbf{R}^n$, of a vector atom f to two vector arguments is

$$\tilde{f}(x, y) = \left(f\left(\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \right), f\left(\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \right), \ldots, f\left(\begin{bmatrix} x_n \\ y_n \end{bmatrix} \right) \right).$$

The extension to an arbitrary number of input arguments can be obtained inductively. Whenever vector functions are called with variable arguments, this extension is automatically applied.

As an example, the expression `norm(x,y,z)` forms the vector expression

$$(\|(x_1, y_1, z_1)\|_2, \|(x_2, y_2, z_2)\|_2),$$

where $x = (x_1, x_2)$, $y = (y_1, y_2)$, and $z = (z_1, z_2)$. The extension can be thought of as first performing a horizontal concatenation and applying the atom *row-wise*.

3.4 Expressions

Expressions in `QCML` are formed by applying basic operators and functions to numeric constants and to previously declared `parameters` and `variables`. Every expression has additional properties:

- a *shape* (either scalar, vector, or matrix),
- a *sign* (either positive, negative, or neither),
- and a *curvature* (either convex, concave, affine, or nonconvex).

An expression with *nonconvex* curvature only means that it cannot be verified as either convex, concave, or affine. The shape and sign of the expression is inferred starting from the properties of the basic objects and applying the operators and atoms (as listed in Table 1 and Table 2). The expression curvature is inferred in a similar way but using disciplined convex programming (DCP) rules. We detail the DCP rules in §4.

3.5 Objectives

Objectives are optional in `QCML`. If no objective is provided, the problem is assumed to be a feasibility problem. A (nonempty) objective consists of a *sense*, either `minimize` or `maximize` and a scalar objective expression.

The objective expression must agree with the objective sense. If the sense is `minimize`, then the expression must be convex (or affine). If the sense if `maximize`, then the expression must be concave (or affine).

3.6 Constraints

Constraints are formed using the typical relational operators: `<=`, `>=`, and `==`. The only restriction for these operators is the curvature of the lefthand and righthand sides. Valid constraints are:

- affine expression `==` affine expression
- convex expression `<=` concave expression
- concave expression `>=` convex expression

Note that affine expressions are both convex and concave, so can be used in either side of an inequality constraint.

3.7 Problems

A `QCML` problem consists of a list of basic type declarations, zero or one objective, and zero or more constraints. Here is a `QCML` problem for completeness:

```
dimensions m n
variable x(n)
parameter mu(n)
parameter gamma positive
parameter F(n,m)
parameter D(n,n)

maximize (mu'*x - gamma*(square(norm(F'*x)) + square(norm(D*x))))
subject to
    sum(x) == 1
    x >= 0
```

This sample problem has six type declarations, one maximization objective, and two constraints. The `subject to` keyword is optional.

4 Disciplined convex programming

Disciplined convex programming describes how to infer the curvature of an expression using only local information (from the immediate operands of an operation) [3]. We start from the leaves of an expression, which consists of basic objects, `variables`, `parameters`, and numeric constants.

All three of these have *affine* curvature. We then infer the curvature of the expression by recursively applying the following composition rule: $\phi(g_1(x), \dots, g_n(x))$ is convex if the atom ϕ is convex and for each i

- g_i is affine, or
- ϕ is increasing in the ith argument and g_i is convex, or
- ϕ is decreasing in the ith argument and g_i is concave.

The rule for verifying concavity is similar. The expression is affine if ϕ is affine and $g_1, \dots, g_n$ are all affine.

5 Abstract dimensions

Dimensions are initially specified as abstract values and can be left abstract in `QCML`. These abstract values must be converted into concrete values before the problem can be solved. There are two ways to make dimensions concrete:

1. specified prior to code generation
2. specified after code generation by passing a `dims` map (as a dictionary or struct) to the generated functions

Dimensions can be set prior to code generation by supplying a partial map, *i.e.*, if `problem` contains a `QCML` problem with dimensions `m` and `n`, then the Python code

```
>>> p = QCML()
>>> p.parse(problem)
>>> p.dims = {'m':  5}
```

will set the dimension `m` to 5 in all expressions, but leave the dimension `n` to be specified later.

Any dimensions specified before code generation will be hard-coded into the resulting problem formulation functions. Thus all problem data fed into the generated code must match these prespecified dimensions. In other words, the generated code is *specialized* to this fixed dimension.

Dimensions that are left abstract allow users to specify problems of differing size, but the dimensions of the input problem must be supplied at the same time. A future release may allow some dimensions to be inferred from the size of the inputs.

6 Code generation

Once a `QCML` problem has been parsed and canonicalized, code is generated to perform the matrix stuffing, converting the parameters in an optimization problem into the data for a standard cone program. This is done by the `codegen` function. The `save` function saves the generated code to a folder or file for use later. The generated code can be used with `QCML` as a dependency and thus can be easily included with any project.

The `codegen` function generates two functions in the desired language:

- a `prob_to_socp` function which maps `parameters` into the cone program data, and

- a `socp_to_prob` function which unpacks a solution of the cone program into the `variable`s of the problem.

We describe specifics of these functions in the sequel.

6.1 Python

To generate Python code, invoke the `codegen` function with the string `'python'` as the argument. This will generate two functions

- `prob_to_socp(params, dims)` and
- `socp_to_prob(x, dims)`.

Invoking the `save` function will save the two functions to a folder.

6.1.1 The `prob_to_socp` function

The `prob_to_socp` function takes two dictionary arguments, a `params` and a `dims` dictionary. If all dimensions are concrete before `codegen` is invoked, the `dims` argument is optional. The keys in these dictionaries correspond to the names of the `parameter`s and the remaining (abstract) `dimension`s specified in the problem string.

The `dims` values must be Python integers. Vector `parameter`s are supplied by Numpy arrays (not matrices), and matrix `parameter`s can be either Numpy arrays or Scipy sparse matrices.

Any extra parameters or dimensions will be unused.

The function will return a dictionary which contains all the information necessary for solving a cone program. This consists of the key-value pairs

- `c`, a Numpy array,
- `G`, a Scipy sparse matrix in column-compressed format,
- `h`, a Numpy array,
- `A`, a Scipy sparse matrix in column-compressed format,
- `b`, a Numpy array, and
- `dims`, a Python dictionary with the key-value pairs,
  - `l`, the number of linear cones, and
  - `q`, a list of the size of second-order cones.

The `dims` output dictionary, which describes the cones, is not to be confused with the `dims` input dictionary, which specifies problem dimensions. The ouput dictionary corresponds to the problem data in (1) and are sufficient for *any* conic solver.

The data `G` and `h` may be 'None,' in which case the standard form `QCML` problem has no conic constraints. Similarly, `A` and `b` may be 'None,' in which case the standard form `QCML` problem has no equality constraints.

6.1.2 The `socp_to_prob` function

The `socp_to_prob` function takes two arguments, `x` and `dims`. The `dims` argument is the same as in `prob_to_socp`; the `x` argument is the solution to the cone program and returns a dictionary with keys corresponding to the names of the `variable`s in the given `QCML` problem. The values of these entries are Numpy arrays (if vectors) or Python floats.

The input argument is a Numpy array.

6.1.3 Dynamic execution

In addition to saving the generated Python source code, the code is compiled into Python bytecode during runtime and can be *executed* right after `codegen`. After the `codegen` step, the QCML object will have two new methods, `prob2socp` and `socp2prob`, which correspond to the generated code. These methods can be used without having to save the Python source.

6.2 Matlab

To generate Matlab code, invoke the `codegen` function with the string `'matlab'` as the argument. This will generate two functions

- `prob_to_socp(params, dims)` and
- `socp_to_prob(x, dims)`.

Invoking the `save` function will save the two functions to a folder.

6.2.1 The `prob_to_socp` function

The `prob_to_socp` function takes two Matlab struct arguments, a `params` and a `dims` dictionary. If all dimensions are concrete before `codegen` is invoked, the `dims` argument is unused. The fields in these structs correspond to the names of the `parameter`s and the remaining (abstract) `dimensions` specified in the problem string.

Any extra parameters or dimensions will be unused.

The function will return a struct which contains all the information necessary for solving a cone program. This consists of the fields

- `c`, a dense vector,
- `G`, a sparse matrix,
- `h`, a dense vector,
- `A`, a sparse matrix,
- `b`, a dense vector, and
- `dims`, a Matlab struct with the fields,
  - `l`, the number of linear cones, and
  - `q`, a vector with the size of second-order cones.

The `dims` output struct, which describes the cones, is not to be confused with the `dims` input struct, which specifies problem dimensions. The ouput struct corresponds to the problem data in (1) and are sufficient for *any* conic solver.

The matrix `G` and vector `h` may be empty, in which case the standard form `QCML` problem has no conic constraints. Similarly, `A` and `b` may be empty, in which case the standard form `QCML` problem has no equality constraints.

6.2.2 The `socp_to_prob` function

The `socp_to_prob` function takes two arguments, `x` and `dims`. The `dims` argument is the same as in `prob_to_socp`; the `x` argument is the solution to the cone program and returns a struct with fields corresponding to the names of the `variable`s in the given `QCML` problem.

6.3 C

To generate C code, invoke the `codegen` function with the string `'C'` as the argument. This will generate two functions

- `qc_socp *qc_prob2socp(const prob_params *params, const prob_dims *dims)`, and
- `void qc_socp2prob(double *x, prob_vars *vars, const prob_dims *dims)`.

Invoking the `save` function will save the two functions to a folder along with several other files:

- a `qcml_utils.h` header file defining matrix and `qc_socp` data structures,
- a `qcml_utils.c` source file implementing matrix utility functions,
- a `prob.h` header file defining `qc_prob2socp` and `qc_socp2prob`,
- a `prob.c` source file implementing the matrix stuffing, and
- a `Makefile` to compile the sources into object files.

To prevent naming clashes, occurences of `prob` in filenames, function names, and struct names are replaced by the name of the folder specified in the `save` method.

The `qcml_utils` header and source file are *not* custom generated for each problem. Instead, they contain the common matrix data structures and functions needed by *any* generated code. Thus, multiple problems can use the *same* `qcml_utils` header and source code.

The supplied `Makefile` only compiles the source code into binary objects. It is up to the user to link them against a main binary.

6.3.1 The `prob_to_socp` function

The `prob_to_socp` function takes two struct arguments, a `prob_params` and a `prob_dims` struct. If all dimensions are concrete before `codegen` is invoked, the `prob_dims` argument can be `NULL`. The fields in these structus correspond to the names of the `parameter`s and the remaining (abstract) `dimension`s specified in the problem string.

Vector `parameter`s are supplied as a pointer to (dense) double arrays, and matrix `parameter`s are supplied as a pointer to the custom `qc_matrix` struct, which stores the matrix in triplet (i, j, v) format.

Any extra parameters or dimensions will be unused.

The function will return a `qc_socp` struct which contains all the information necessary for solving a cone program. This consists of the fields (with their type):

- `long n`, the number of variables in (1),
- `long m`, number of cone constraints in (1),
- `long p`, number of equality constraints in (1),
- `long l`, number of linear cones,
- `long nsoc`, number of second-order cones,
- `long *q`, list of second-order cone sizes,
- `double *G`, nonzero values of G (in column-compressed format),
- `long *Gp`, column pointers of G (in column-compressed format),
- `long *Gi`, row values of G (in column-compressed format),
- `double *A`, nonzero values of A (in column-compressed format),
- `long *Ap`, column pointers of A (in column-compressed format),
- `long *Ai`, row values of A (in column-compressed format),
- `double *c`, c vector (dense),
- `double *h`, h vector (dense), and
- `double *b`, b vector (dense),

The `qc_socp` struct corresponds to the problem data in (1) and are sufficient for *any* conic solver. It must be freed when no longer in use.

The data pointers `Gx`, `Gp`, `Gi`, and `h` may be `NULL`, in which case the standard form `QCML` problem has no conic constraints. Similarly, `Ax`, `Ap`, `Ai`, and `b` may be `NULL`, in which case the standard form `QCML` problem has no equality constraints.

6.3.2 The `socp_to_prob` function

The `socp_to_prob` function takes three arguments: a `double *`, which points to a double array containing the solution to the cone program; a `prob_vars` struct, which is set to the *pointers* in the underlying `prob_vars` struct to the `variable`s in the `QCML` problem; and a `prob_dims` struct.

7 Rapid prototyping

`QCML` also provides a helper function for rapid prototyping in Python. This is useful to verify the optimization model before generating code to use elsewhere. The `solve` function exectues the `parse`, `canonicalize`, and `codegen` functions in succession. It also takes the generated source code and produces bytecode that is executed on-the-fly.

Thus, if `prob` is a `QCML` problem, then

```
>>> p = QCML()
>>> p.parse(prob)
>>> p.solve()
```

will use parameters and dimensions defined locally to stuff the matrices and invoke the ECOS solver to solve the problem.

This feature is useful in Python to test (small) models before commiting to a source code.

8 Lasso example

In this section, we will walk through a simple `QCML` use-case by implementing a lasso (ℓ_1-regularized least squares) solver in C. This example and others like it can be found under the `qcml/examples` directory.

The lasso problem is

$$\text{minimize} \quad \|Ax - b\|_2^2 + \lambda\|x\|_1,$$

with variable $x \in \mathbf{R}^n$ and problem data $A \in \mathbf{R}^{m\times n}$, $b \in \mathbf{R}^m$, and $\lambda > 0$. We can solve this problem using a standard interior-point solver, such as ECOS, with the transformation to standard cone form provided by `QCML`.

First, we begin by generating some random problem data using Numpy,

```
from numpy.random import randn
n = 1000 # number of features
m = 100 # number of examples
A = randn(m,n)
b = randn(m)
gamma = 1
```

We then specify the lasso problem in `QCML` as follows:

```
dimensions m n
variable x(n)
parameters A(m,n) b(m)
parameter gamma positive

minimize square(norm(A*x - b)) + gamma*norm1(x)
```

Storing this description in the Python string variable `prob`, we first solve the problem using ECOS,

```
from qcml import QCML
p = QCML()
p.parse(prob)
res = p.solve()
```

Typically, at this stage, one would tweak the model as needed to obtain better performance (on a cross-validation set, say), but since we have committed to solving the lasso problem, this step is only to verify that the problem is properly solved.

We can print the value of `res['objval']` to see the optimal value of the randomly generated lasso problem and save the randomly generated data for validation later. Next, we generate C source code for the lasso problem

```
p.codegen("C")
p.save("lasso")
```

This will create a directory named '`lasso`' which contains the source code along with a Makefile. The generated functions will be named `qc_lasso2socp` and `qc_socp2lasso`. We can then write a separate main function which calls the `qc_lasso2socp` to create the necessary data structures for calling the ECOS solver:

```
/* read Python-generated data into arrays */
double *Av = ...; /* nonzero values of A parameter */
double *b = ...; /* values of b parameter */
long *Ai = ...; /* nonzero row indices */
long *Aj = ...; /* nonzero col indices */

/* create parameter, dimension, and solution structs */
lasso_params p;
lasso_dims d;
lasso_vars sol;

/* matrix used in lasso_params */
qc_matrix A;
A.v = Av; A.i = Ai; A.j = Aj; A.nnz = m*n;
A.m = m; A.n = n; /* m, n provided exogenously */

/* set the parameters and dims*/
p.A = &A; p.b = b; p.gamma = 1.0;
d.m = A.m; d.n = A.n;

/* stuff the matrices */
qc_socp *data = qc_lasso2socp(&p, &d);

/* run ecos and solve it */
pwork *mywork = ECOS_setup(data->n, data->m, data->p,
  data->l, data->nsoc, data->q,
  data->Gx, data->Gp, data->Gi,
  data->Ax, data->Ap, data->Ai,
  data->c, data->h, data->b);
if (mywork)
{
  ECOS_solve(mywork);

  /* recover the solution */
  qc_socp2lasso(mywork->x, &sol, &d);
  ...  sol.x ...  /* do something with solution */
}

/* free the data */
qc_socp_free(data);
if (mywork) ECOS_cleanup(mywork, 0);
```

Finally, the main file can be linked against the `qcml_utils` and generated `lasso` objects and used as a stand-alone binary.

References

[1] E. Chu, N. Parikh, A. Domahidi, and S. Boyd. Code generation for embedded second-order cone programming. In *European Control Conference*, pages 1547–1552, July 2013.

[2] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer, 2008.

[3] M. Grant, S. Boyd, and Y. Ye. CVX: Matlab software for disciplined convex programming, ver. 2.0, build 870. Available at `www.stanford.edu/~boyd/cvx/`, September 2012.