Here are my thoughts on what’s done well in CVXPY, what’s missing, and what I’d like to see in a core CVX library.

1 What’s done well in CVXPY

The best design decision I made in CVXPY was to view everything as an expression tree. When the user types in a mathematical expression like \( \text{abs}(x) + 2x + 3 \) it’s converted via operator overloading into an expression tree, as depicted in Figure 1.

The leaves of an expression tree represent variables, constants, or parameters (more about these later). The non-leaf nodes represent functions. Notice that addition is treated as just another function. Even operations like indexing and transpose are viewed as functions and represented as nodes in the expression tree.

During canonicalization, the problem is rewritten as an objective represented by an affine expression tree (i.e., containing only linear functions) and a list of cone constraints, also represented as affine expression trees. I say affine rather than linear because the leaves may be constants.
During matrix stuffing, each function in an affine expression tree is replaced with its matrix representation, and the tree is flattened into a map of variable id to coefficient.

The affine expression tree intermediate representation makes it easy to handle parameters. Parameters are constants with fixed sign and shape, but whose value can change. If you solve a parameterized problem multiple times with different parameter values, most of the work of canonicalization and matrix stuffing will only happen during the first solve. Since canonicalization doesn’t look at the values of constants, you can just plug parameters into the canonicalization algorithm. CVXPY caches the entries of the cone program matrix corresponding to expression trees without parameters. Expression trees with parameters are reprocessed during each solve. It would be easy to implement a more sophisticated approach that introduces new variables so that parameters map to a fixed block in the cone program matrix.

Another design decision that worked out well in CVXPY was to allow free floating variables and expressions that aren’t connected to any particular problem. This allows alternative approaches to expressing optimization problems. For example, I’m working on a project to integrate CVXPY with the graph analytics library SNAP. The idea is you construct a graph where each node and each edge has a cost function and constraints.

2 What’s missing in CVXPY

The biggest thing missing in CVXPY is a presolve. There’s a very minimal presolve, but generally I’m expecting the solvers to handle redundant constraints. ECOS and SCS do fine, but there were a lot of issues in taking this approach with CVXOPT. It would be better to remove redundant constraints and simplify the problem in general.

Also, CVXPY always solves the primal problem. I gather that CVX tries to be more efficient by sometimes solving the dual problem. Does that make a big difference? I’d be interested to hear more about how CVX approaches this.

3 What I’d like in the CVX core

I want the central CVX library to be very modular. We would have a limited set of data structures, but many modules that operate on those data structures.

3.1 Data structures

One important data structure would be expression trees (surprise!).

Each node in an expression tree has the following attributes:

- The name of the function (or some symbol representing the function).
- The dimensions of the function output.
• Other data needed to parameterize the function, such as the matrix for matrix multiplication, the index for indexing, the power \( p \) for \( x^p \).

• A list of arguments to the function. These are the node’s children.

It might actually be better to have an expression DAG, so that you can have multiple output functions like \( \text{copy} : \mathbb{C}^n \to \mathbb{C}^n \times \ldots \times \mathbb{C}^n \), which outputs many replicas of its input. In that case the node would also have a list of outputs, i.e., nodes it sends outputs to.

Another data structure would be constraints, which are an expression tree whose root represents membership in a set. Set membership is a nice way to capture equality, inequality, and generalized (conic) inequality constraints. It also allows nonconic constraints, which we might want at some point.

Problems are a tuple \((s, o, C)\) where \( s \) is an objective sense (minimize or maximize), \( o \) is an expression tree representing the objective, and \( C \) is a list or set of constraints.

Another data structure is a cone program representation, which is the cone program expressed with matrices. I don’t care so much what standard form we use. The important thing is that we have an internal standard form that we later convert into the standard form for a particular solver. That way canonicalization and matrix stuffing are solver independent.

I suggest we use the MathProgBase standard form and data structure (described here: http://mathprogbasejl.readthedocs.org/en/latest/conic.html). The standard form is

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad b - Ax \in K_1 \\
& \quad x \in K_2,
\end{align*}
\]

where \( x \in \mathbb{R}^n \) is the optimization variable, \( A \in \mathbb{R}^{m \times n} \), \( b \in \mathbb{R}^m \), and \( c \in \mathbb{R}^n \) are problem data, and \( K_1 \) and \( K_2 \) are products of convex cones. I gather that some of the solvers CVX accesses can handle complex variables and data natively, so we might want a complex standard form instead.

The data structure is specified by a sparse matrix \( A \), vectors \( b \) and \( c \), and a list of (cone symbol, range) tuples which indicate that a given section of \( K_1 \) corresponds to the specified cone. There is a similar list for \( K_2 \).

We would also need a version of the cone program representation that incorporated parameters. I would recommend the same data structure plus a list of (expression tree, block selector) tuples. The expression trees would only have parameters as leaves. Each tuple maps a function of the parameters (represented by the expression tree) to a block of the matrix \( A \). We would have similar lists mapping expression trees to slices of \( b \).

We would need another variant that supported integer and boolean constraints. The way these are expressed in CVXPY is similar to the \( K_2 \) constraints. Certain indices/slices of the variable are specified as boolean or integer.

### 3.2 Modules

Here are some of the modules we would have:
The “DCP check” module would operate on problems, expression trees, and constraints. The module would take in one of these objects and a map of function name to information about the function, such as convexity and monotonicity in each argument, and return the DCP determined convexity. If it’s doing signed DCP, it would also return the sign (and require sign information about the functions). I think it’s better to keep this information separate rather than put it in the expression tree data structure. The expression tree nodes should have just enough information to define what the function is. That makes the data structure more general. Some functions have a fixed output dimension, but so many don’t that it’s worth including it for all.

The “Canonicalize” module would map problems to problems. The module expands functions into their cone program representations. It returns a problem where all the expression trees contain only linear functions.

We may want “Canonicalize” to also take in a list of supported cones. Another approach would be to have canonicalization try and use the best/most specific cones possible and have another module that rewrites the cone program in terms of the cones the solver actually supports.

The “Matrix stuff” module takes a problem with all affine expression trees and returns a cone program representation and some information mapping variables to columns and constraints to rows.

We would have presolve and dualize modules. These take in a cone program representation and return a cone program representation.

We would have modules for each solver that convert our cone program representation into the standard form required by the solver.

We would also have modules that take the results from the solver and repackaging them in terms of our cone program standard form. Similarly, we would have a module that took the solution and the mapping of variables to columns and constraints to rows and unpacked values of the primal and dual variables.

The whole pipeline would look like this:

|DCP check| => |Canonicalize| => |Matrix stuff| => |Presolve/Dualize| => |Convert to solver standard form| => |Solver| => |Extract results in standard form| => |Map solution onto variables and constraints (for duals)|

We would at some point want modules that simplify expression trees. These could go before and after canonicalize.

### 3.3 Other thoughts

The benefit of having a few well defined data structures and many modules is that we can swap in new modules that do very different things without any difficulty.

For example, we might want to add a module that canonicalizes into POGS standard form (i.e., graph ADMM form) instead of cone program standard form. This is going to require a fairly different approach to canonicalization, and different information about functions (e.g., which have a prox operator).
Another example is a differentiation module. This would be very useful for solving sequential convex programs. The expression tree data structure could be easily used by such a module because it’s just specifying function composition. Ideally some existing differentiation tool would make an interface that took our data structure. That’s the sort of thing I had in mind when I said that we want to make a symbolic math intermediate representation that everyone can use.