Symbolic Subdifferentiation in Python

Maurizio Caló and Jaehyun Park
EE 364B Project Final Report
Stanford University, Spring 2010-11

June 2, 2011

1 Introduction

1.1 Subgradient-PY

We implemented a Python package `Subgradient-PY` (`spy`) that solves optimization problems using subgradient methods. Below are some features of `spy`.

- The objective function and constraints need not be differentiable.
- Due to little space usage, `spy` can be used to solve large problems.
- `spy` can be combined with primal or dual decomposition techniques to develop distributed algorithms.

The essential part of the project is to compute subgradients correctly and efficiently.

1.2 Motivation

This project is motivated by the first two weeks of EE 364B lectures on subgradients and subgradient methods. Also, somewhat unexpectedly, there is yet no software package that can automatically compute subgradients of convex functions, or implement subgradient methods. (S. Boyd, personal communication, April 2011) Therefore, we decided to implement an automatic subgradient calculator, and extend it to a convex optimization problem solver based on subgradient methods.

`spy` is implemented in Python. The objected-oriented paradigm of Python made it easy to implement a nontrivial project in a short time, without worrying too much about technical details. There has been some attempts to model convex optimization problems in Python, such as `cvxpy`, `cvxmod`, and `cvxopt`. However, rather than integrating our project with these existing softwares, we decided to implement everything from scratch, for educational purposes.

2 A Quick Start

`spy` can be downloaded from `http://www.stanford.edu/~liszt90/spy`. There is no installation package, so one can simply download the codes and unzip it to a directory named `spy`. The zipped file also contains example codes, as well as all the unit tests.

2.1 Example Code

The following code solves a simple convex optimization problem using `spy`.

```
from spy import *
x = var('x')
y = var('y')
ex = max(x + y, 2 * x - y) + square(x)
constraints = [geq(x, y), leq(norm2([x, y]), 1)]
prob = minimize(ex, constraints)
(optval, optpoint) = prob.solve()
```

Let us examine the code line by line:

- Line 1: Import all modules from the `spy` package.
- Line 2-3: Declare x and y as (scalar) optimization variables.
- Line 4: Form a symbolic expression using the variables declared in the previous line. The expression corresponds to the mathematical expression $\max(x+y, 2x-y)+x^2$.
- Line 5: Specify the constraints of the optimization problem: $x \geq y$ and $\sqrt{x^2+y^2} \leq 1$.
- Line 6: Define the optimization problem: "minimize `ex` subject to `constraints`."
- Line 7: Solve the optimization problem using a subgradient method.

Printing out `optval` and `optpoint` gives the following:

```
-0.250040107722
{'y': -0.50654454816398864, 'x': -0.50662859414656536}
```

The analytic solution is $x = y = -1/2$ with the optimal value $f^\star = -1/4$. The accuracy of the solution can be improved by adjusting parameters of the subgradient method. For more information, refer to Section 4.3.

3 Basics

3.1 Main Classes

`spy` consists of three main components:

- Expression: Any real-valued mathematical expression is an object of Expression type. It can contain variables whose values are not predetermined. For convex expressions that follow the DCP ruleset (refer to Section 4.2), subgradients can be evaluated at a given point.
- Constraint: A constraint is an inequality of the form (convex) $\leq$ (concave) or an equality of the form (affine) = (affine). Each constraint represents a set of values that the optimization variables can take.
- (Optimization) Problem: A problem is a triplet of the form (minimize/maximize, objective, list of constraints). The objective function of the minimization (resp. maximization) problem must be convex (resp. concave).

3.2 Declaring Variables

One can declare a variable using the following command.

```
x = var('x')
```

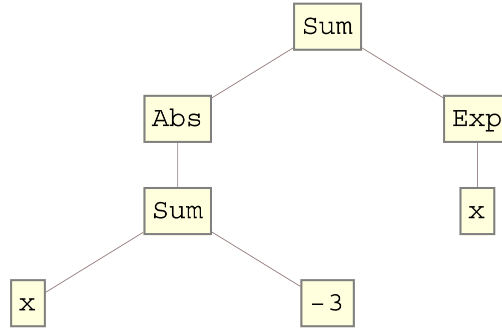
The meaning should be clear from the syntax; the line above declares a variable `'x'`, whose value is not predetermined. The object `x` is a symbolic link to the variable with the identifier string `'x'`. Different variable objects with the same identifier string are recognized as the same variable by `spy`. This design allows users to recover variables from identifier strings even if they lose variable objects.

3.3 Forming an Expression

With variables, it is possible to form more complicated expressions. To provide a clean interface to the user, we have overloaded basic arithmetic operators as well as some of the Python built-in functions. This allows users to form expressions in a natural way. For instance, the following line will create an expression named `ex`, using a variable object `x`.

```
ex = abs(x - 3) + exp(x)
```

This expression corresponds to a mathematical expression $|x - 3| + e^x$. It should be noted that `spy` does not attempt to simplify an expression, nor does it immediately compute the value of a newly formed expression. All computations are done lazily; the values will be computed only when the `get_value` method is called on it. Internally, the expression is stored as a tree shown in the following diagram:



3.4 Retrieving Values or Subgradients

Because the values of variables in an expression are not predetermined, a user needs to specify them manually in order to retrieve the actual value or a subgradient of the expression. The values of the variables are not passed in as a list of numbers as one might expect. Rather, the values are passed as a dictionary that maps the name of variables to their values. The benefit of this design is that one can readily access the value of a variable by its name.

```
varmap = {'x': 1.5, 'y': -2}
val = ex.get_value(varmap)
g = ex.subgrad(varmap)
```

The code above will evaluate the expression `ex` at $x = 1.5$ and $y = -2$, and store the result in `val`. Note that the argument to `get_value` is a dictionary, not a list of coordinates. The next line computes a subgradient of `ex` evaluated at the same point, storing the result into `g`. The data type of `g` is the same as that of `varmap`. When there are multiple subgradients at the given point, `spy` will try to give different subgradients every time `get_value` is called. For these methods to work correctly, users need to specify the values of all the variables present in the expression. Otherwise, `nan` will be returned. Users can retrieve the list of variables in an expression by calling `get_vars` method on it.

The subgradient computation, as mentioned above, is the most crucial part of this project. This feature has been tested by multiple scripts that solve simple optimization problems. Also, whenever a new library function is added, a unit test is implemented in order to ensure that the functions behave as expected.

3.5 Specifying Constraints

One can use `leq`, `eq`, or `geq` to construct a constraint object. At construction time, `spy` automatically checks whether inequalities and equalities follow the DCP ruleset.

Another important feature of the Constraint class is `cutting_plane` function. This method returns a deep cutting plane whenever a constraint is violated. Let f be a convex function, and assume that $f(x) > 0$ for some x. For a point z to satisfy $f(z) \leq 0$, it must satisfy the inequality given by the deep cutting plane: $g^T z + f(x) - g^T x < 0$. Here, $g \in \partial f(x)$.

3.6 Solving Optimization Problems

One can describe and solve a convex optimization problem using the following syntax:

```
myprob = minimize(ex, [cons1, cons2])
(optval, optpoint) = myprob.solve()
```

Here, `myprob` is a description of an optimization problem. The second line calls `solve` method, which minimizes the expression `ex` subject to two constraints `cons1` and `cons2` using the subgradient method. As a result, the optimal point as well as the objective value at that point will be returned. Advanced users can specify the step size rule used by the method. Refer to Section 4.3.

4 Technical Details

4.1 Computation of Subgradients

Subgradient computations are done recursively, and the procedure is analogous to the chain rule for differentiable functions. For each library function, we've implemented a method that computes its subgradient at a given point. For the complete list of library functions, see section 4.4.

Subgradients of complex expressions can be computed using the composition rule [1]:

- Let $f(x) = h(f_1(x), \dots, f_k(x))$, with h convex non-decreasing, each f_i convex.
- Find $c \in \partial h(f_1(x), \dots, f_k(x))$, $g_i \in \partial f_i(x)$.
- Then, $g = c_1 g_1 + c_2 g_2 + \cdots + c_k g_k$ is a subgradient of f at x.

4.2 DCP Ruleset

Whenever a Constraint object is constructed, `spy` validates the convexity of it using the DCP ruleset [2]. Similarly, when a Problem object is formed, `spy` automatically checks whether the objective of the minimization (resp. maximization) is convex (resp. concave). This is done by applying the general vector composition rule [3], using the monotonicity and convexity properties of the atomic functions.

4.3 Subgradient Method

The `solve` method of the Problem class implements a subgradient method, which is similar to gradient descent methods for minimizing differentiable functions. Consider the following optimization problem.

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \dots, m. \end{array}$$

By default, the `solve` method sets every variable in f as zero, and repeatedly applies the following update rule until convergence, or until the iteration limit is reached. Related constants can be adjusted in `constants.py`.

- Let $x^{(k)}$ be the kth iteration point. If $x^{(k)}$ is feasible, find $g^{(k)} \in \partial f(x^{(k)})$. Otherwise, find the constraint that is most violated, say f_i, and find $g^{(k)} \in \partial f_i(x^{(k)})$.
- Set the next point as $x^{(k+1)} := x^{(k)} - \alpha_k g^{(k)}$, where α_k is kth step size.

If the step size rule is not specified, `spy` uses $\alpha_k = 1/k$ as the default step size. Custom step size rules can be used as well, using a syntax like the following:

```
stepsize = (var('f') - f_estimate) / square(var('gnorm'))
prob.solve(stepsize)
```

The argument that is passed to the `solve` method can be any expression of three variables, `f`, `gnorm`, and `iter`, which correspond to $f^{(k)}$, $\left\|g^{(k)}\right\|_2$, and k, respectively. The step size must evaluate to a positive number during the execution of the `solve` method.

Finally, one can specify the initial point $x^{(1)}$ as an additional argument to `solve`. In this case, the step size rule must be provided as well.

4.4 Library Functions

The following list contains the functions that are implemented in the current version of `spy`. For more details, please refer to the `cvx` user guide [2].

- `abs`, `max`, `min`, `pos`, `power`, `power_pos`
- `exp`, `log`, `log_sum_exp`, `rel_entr`
- `norm1`, `norm2`, `berhu`, `huber`
- `square`, `square_pos`, `sqrt`, `geo_mean`, `quad_over_lin`

5 Limitations

We are aware of the following issues with `spy`.

- `spy` does not work well with equality constraints yet.
- Convergence behavior depends on the step sizes, so it is up to users to give a “right” step size rule.
- Currently, `spy` cannot detect if the given problem is unbounded.
- `spy` only works with scalar-valued variables and expressions. We have a simple implementation of the matrix class, but it is not yet integrated with `spy`. This class is written from scratch as well, and does not depend on any other scientific computing packages such as `numpy`.

References

[1] S. Boyd, *Subgradients*, EE 364B Lecture Slides, 2011.

[2] M. Grant and S. Boyd, `cvx` *Users' Guide*, 2011.

[3] S. Boyd, *Convex Functions*, EE 364A Lecture Slides, 2011.