Convex.jl Documentation

Release 0.1

Karanveer Mohan, Madeleine Udell, David Zeng, Jenny Hong, Stev

October 15, 2015

1 In Depth Documentation: 3
1.1 Installation . . . 3
1.2 Quick Tutorial . . . 3
1.3 Basic Types . . . 4
1.4 Operations . . . 6
1.5 Examples . . . 7
1.6 Solvers . . . 7
1.7 FAQ . . . 8
1.8 Advanced Features . . . 10
1.9 Credits . . . 11

Convex.jl is a Julia package for Disciplined Convex Programming (DCP). Convex.jl makes it easy to describe optimization problems in a natural, mathematical syntax, and to solve those problems using a variety of different (commercial and open-source) solvers. Convex.jl can solve

- linear programs
- mixed-integer linear programs and mixed-integer second-order cone programs
- dcp-compliant convex programs including
  - second-order cone programs (SOCP)
  - exponential cone programs
  - semidefinite programs (SDP)

Convex.jl supports many solvers, including Mosek, Gurobi, ECOS, SCS and GLPK, through the MathProgBase interface.

Note that Convex.jl was previously called CVX.jl. This package is under active development; we welcome bug reports and feature requests. For usage questions, please contact us via the JuliaOpt mailing list.

CHAPTER 1

In Depth Documentation:

1.1 Installation

Installing Convex.jl is a one step process. Open up Julia and type

```
Pkg.update()
Pkg.add("Convex")
```

This does not install any solvers. If you don't have a solver installed already, you will want to install a solver such as SCS by running

```
Pkg.add("SCS")
```

To solve certain problems such as mixed integer programming problems you will need to install another solver as well, such as GLPK. If you wish to use other solvers, please read the section on solvers.

1.2 Quick Tutorial

Consider a constrained least squares problem

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 \\ \text{subject to} & x \geq 0 \end{array}$$

with variable $x \in \mathbf{R}^n$, and problem data $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$.

This problem can be solved in Convex.jl as follows:

```
# Make the Convex.jl module available
using Convex

# Generate random problem data
m = 4;  n = 5
A = randn(m, n); b = randn(m, 1)

# Create a (column vector) variable of size n x 1.
x = Variable(n)

# The problem is to minimize ||Ax - b||^2 subject to x >= 0
# This can be done by: minimize(objective, constraints)
problem = minimize(sumsquares(A * x - b), [x >= 0])
```

```
# Solve the problem by calling solve!
solve!(problem)

# Check the status of the problem
problem.status # :Optimal, :Infeasible, :Unbounded etc.

# Get the optimum value
problem.optval
```

1.3 Basic Types

The basic building block of Convex.jl is called an *expression*, which can represent a variable, a constant, or a function of another expression. We discuss each kind of expression in turn.

1.3.1 Variables

The simplest kind of expression in Convex.jl is a variable. Variables in Convex.jl are declared using the *Variable* keyword, along with the dimensions of the variable.

```
# Scalar variable
x = Variable()

# Column vector variable
x = Variable(5)

# Matrix variable
x = Variable(4, 6)
```

Variables may also be declared as having special properties, such as being

- (entrywise) positive: `x = Variable(4, Positive())`
- (entrywise) negative: `x = Variable(4, Negative())`
- integral: `x = Variable(4, :Int)`
- binary: `x = Variable(4, :Bin)`
- (for a matrix) being symmetric, with nonnegative eigenvalues (ie, positive semidefinite): `z = Semidefinite(4)`

1.3.2 Constants

Numbers, vectors, and matrices present in the Julia environment are wrapped automatically into a *Constant* expression when used in a Convex.jl expression.

1.3.3 Expressions

Expressions in Convex.jl are formed by applying any *atom* (mathematical function defined in Convex.jl) to variables, constants, and other expressions. For a list of these functions, see operations. Atoms are applied to expressions using operator overloading. For example, `2+2` calls Julia's built-in addition operator, while `2+x` calls the Convex.jl addition method and returns a Convex.jl expression. Many of the useful language features in Julia, such as arithmetic, array indexing, and matrix transpose are overloaded in Convex.jl so they may be used with variables and expressions just as they are used with native Julia types.

Expressions that are created must be DCP-compliant. More information on DCP can be found here.

```
x = Variable(5)
# The following are all expressions
y = sum(x)
z = 4 * x + y
z_1 = z[1]
```

Convex.jl allows the values of the expressions to be evaluated directly.

```
x = Variable()
y = Variable()
z = Variable()
expr = x + y + z
problem = minimize(expr, x >= 1, y >= x, 4 * z >= y)
solve!(problem)

# Once the problem is solved, we can call evaluate() on expr:
evaluate(expr)
```

1.3.4 Constraints

Constraints in Convex.jl are declared using the standard comparison operators <=, >=, and ==. They specify relations that must hold between two expressions. Convex.jl does not distinguish between strict and non-strict inequality constraints.

```
x = Variable(5, 5)
# Equality constraint
constraint = x == 0
# Inequality constraint
constraint = x >= 1
```

Matrices can also be constrained to be positive semidefinite.

```
x = Variable(3, 3)
y = Variable(3, 1)
z = Variable()
# constrain [x y; y' z] to be positive semidefinite
constraint = ([x y; y' z] in :SDP)
# or equivalently,
constraint = ([x y; y' z]  0)
```

1.3.5 Objective

The objective of the problem is a scalar expression to be maximized or minimized by using `maximize` or `minimize` respectively. Feasibility problems can be expressed by either giving a constant as the objective, or using `problem = satisfy(constraints)`.

1.3.6 Problem

A *problem* in Convex.jl consists of a *sense* (minimize, maximize, or satisfy), an *objective* (an expression to which the sense verb is to be applied), and zero or more *constraints* that must be satisfied at the solution. Problems may be constructed as

```
problem = minimize(objective, constraints)
# or
problem = maximize(objective, constraints)
# or
problem = satisfy(constraints)
```

Constraints can be added at any time before the problem is solved.

```
# No constraints given
problem = minimize(objective)
# Add some constraint
problem.constraints += constraint
# Add many more constraints
problem.constraints += [constraint1, constraint2, ...]
```

A problem can be solved by calling `solve!`:

```
solve!(problem)
```

After the problem is solved, `problem.status` records the status returned by the optimization solver, and can be `:Optimal`, `:Infeasible`, `:Unbounded`, `:Indeterminate` or `:Error`. If the status is `:Optimal`, `problem.optval` will record the optimum value of the problem. The optimal value for each variable `x` participating in the problem can be found in `x.value`. The optimal value of an expression can be found by calling the `evaluate()` function on the expression as follows: `evaluate(expr)`.

1.4 Operations

Convex.jl currently supports the following functions. These functions may be composed according to the DCP composition rules to form new convex, concave, or affine expressions. Convex.jl transforms each problem into an equivalent cone program in order to pass the problem to a specialized solver. Depending on the types of functions used in the problem, the conic constraints may include linear, second-order, exponential, or semidefinite constraints, as well as any binary or integer constraints placed on the variables. Below, we list each function available in Convex.jl organized by the (most complex) type of cone used to represent that function, and indicate which solvers may be used to solve problems with those cones. Problems mixing many different conic constraints can be solved by any solver that supports every kind of cone present in the problem.

In the notes column in the tables below, we denote implicit constraints imposed on the arguments to the function by IC, and parameter restrictions that the arguments must obey by PR. (Convex.jl will automatically impose ICs; the user must make sure to satisfy PRs.)

1.4.1 Linear Program Representable Functions

An optimization problem using only these functions can be solved by any LP solver.

1.4.2 Second-Order Cone Representable Functions

An optimization problem using these functions can be solved by any SOCP solver (including ECOS, SCS, Mosek, Gurobi, and CPLEX). Of course, if an optimization problem has both LP and SOCP representable functions, then any solver that can solve both LPs and SOCPs can solve the problem.

1.4.3 Exponential Cone Representable Functions

An optimization problem using these functions can be solved by any exponential cone solver (SCS).

1.4.4 Semidefinite Program Representable Functions

An optimization problem using these functions can be solved by any SDP solver (including SCS and Mosek).

1.4.5 Exponential + SDP representable Functions

An optimization problem using these functions can be solved by any solver that supports exponential constraints *and* semidefinite constraints simultaneously (SCS).

operation	description	vexity	slope	notes
`logdet(x)`	log of determinant of x	concave	increasing	IC: x is positive semidefinite

1.4.6 Promotions

When an atom or constraint is applied to a scalar and a higher dimensional variable, the scalars are promoted. For example, we can do `max(x, 0)` gives an expression with the shape of `x` whose elements are the maximum of the corresponding element of `x` and `0`.

1.5 Examples

- Basic usage
- Control
- Tomography
- Time series
- Section allocation
- Binary knapsack
- Entropy maximization
- Logistic regression

1.6 Solvers

Convex.jl transforms each problem into an equivalent cone program in order to pass the problem to a specialized solver. Depending on the types of functions used in the problem, the conic constraints may include linear, second-order, exponential, or semidefinite constraints, as well as any binary or integer constraints placed on the variables.

By default, Convex.jl does not install any solvers. Many users use the solver SCS, which is able to solve problems with linear, second-order cone constraints (SOCPs), exponential constraints and semidefinite constraints (SDPs). Any other solver in JuliaOpt may also be used, so long as it supports the conic constraints used to represent the problem. Most other solvers in the JuliaOpt ecosystem can be used to solve (mixed integer) linear programs (LPs and MILPs). Mosek and Gurobi can be used to solve SOCPs (even with binary or integer constraints), and Mosek can also solve

SDPs. For up-to-date information about solver capabilities, please see the table here describing which solvers can solve which kind of problems.

Installing these solvers is very simple. Just follow the instructions in the documentation for that solver.

To use a specific solver, you can use the following syntax

```
solve!(p, GurobiSolver())
solve!(p, MosekSolver())
solve!(p, GLPKSolverMIP())
solve!(p, GLPKSolverLP())
solve!(p, ECOSSolver())
solve!(p, SCSSolver())
```

(Of course, the solver must be installed first.) For example, we can use GLPK to solve a MILP

```
using GLPKMathProgInterface
solve!(p, GLPKSolverMIP())
```

You can set or see the current default solver by

```
get_default_solver()
using Gurobi
set_default_solver(GurobiSolver()) # or set_default_solver(SCSSolver(verbose=0))
# Now Gurobi will be used by default as a solver
```

Many of the solvers also allow options to be passed in. More details can be found in each solver's documentation.

For example, if we wish to turn off printing for the SCS solver (ie, run in quiet mode), we can do so by

```
using SCS
solve!(p, SCSSolver(verbose=false))
```

If we wish to increase the maximum number of iterations for ECOS or SCS, we can do so by

```
using ECOS
solve!(p, ECOSSolver(maxit=10000))
using SCS
solve!(p, SCSSolver(max_iters=10000))
```

1.7 FAQ

- *Where can I get help?*
- *How does Convex.jl differ from JuMP?*
- *Where can I learn more about Convex Optimization?*
- *Are there similar packages available in other languages?*
- *How does Convex.jl work?*
- *How do I cite this package?*

1.7.1 Where can I get help?

For usage questions, please contact us via the JuliaOpt mailing list. If you're running into bugs or have feature requests, please use the Github Issue Tracker.

1.7.2 How does Convex.jl differ from JuMP?

Convex.jl and JuMP are both modelling languages for mathematical programming embedded in Julia, and both interface with solvers via the MathProgBase interface, so many of the same solvers are available in both. Convex.jl converts problems to a standard conic form. This approach requires (and certifies) that the problem is convex and DCP compliant, and guarantees global optimality of the resulting solution. JuMP allows nonlinear programming through an interface that learns about functions via their derivatives. This approach is more flexible (for example, you can optimize non-convex functions), but can't guarantee global optimality if your function is not convex, or warn you if you've entered a non-convex formulation.

For linear programming, the difference is more stylistic. JuMP's syntax is scalar-based and similar to AMPL and GAMS making it easy and fast to create constraints by indexing and summation (like `sum{x[i], i=1:numLocation}`). Convex.jl allows (and prioritizes) linear algebraic and functional constructions (like `max(x,y) < A*z`); indexing and summation are also supported in Convex.jl, but are somewhat slower than in JuMP. JuMP also lets you efficiently solve a sequence of problems when new constraints are added or when coefficients are modified, whereas Convex.jl parses the problem again whenever the *solve!* method is called.

1.7.3 Where can I learn more about Convex Optimization?

See the freely available book Convex Optimization by Boyd and Vandenberghe for general background on convex optimization. For help understanding the rules of Disciplined Convex Programming, we recommend this DCP tutorial website.

1.7.4 Are there similar packages available in other languages?

Indeed! You might use CVXPY in Python, or CVX in Matlab.

1.7.5 How does Convex.jl work?

For a detailed discussion of how Convex.jl works, see our paper.

1.7.6 How do I cite this package?

If you use Convex.jl for published work, we encourage you to cite the software using the following BibTeX citation:

```
@article{convexjl,
 title = {Convex Optimization in {J}ulia},
 author ={Udell, Madeleine and Mohan, Karanveer and Zeng, David and Hong, Jenny and Diamond, Steven a
 year = {2014},
 journal = {SC14 Workshop on High Performance Technical Computing in Dynamic Languages},
 archivePrefix = "arXiv",
 eprint = {1410.4821},
 primaryClass = "math-oc",
}
```

1.8 Advanced Features

1.8.1 Dual Variables

Convex.jl also returns the optimal dual variables for a problem. These are stored in the `dual` field associated with each constraint.

```
using Convex

x = Variable()
constraint = x >= 0
p = minimize(x, constraint)
solve!(p)

# Get the dual value for the constraint
p.constraints[1].dual
# or
constraint.dual
```

1.8.2 Warmstarting

If you're solving the same problem many times with different values of a parameter, Convex.jl can initialize many solvers with the solution to the previous problem, which sometimes speeds up the solution time. This is called a **warm start**.

To use this feature, pass the optional argument *warmstart=true* to the *solve!* method.

```
# initialize data
n = 1000
y = rand(n)
x = Variable(n)

# first solve
lambda = 100
problem = minimize(sumsquares(y - x) + lambda * sumsquares(x - 10))
@time solve!(problem)

# now warmstart
# if the solver takes advantage of warmstarts,
# this run will be faster
lambda = 105
@time solve!(problem, warmstart=true)
```

1.8.3 Fixing and freeing variables

Convex.jl allows you to fix a variable *x* to a value by calling the *fix!* method. Fixing the variable essentially turns it into a constant. Fixed variables are sometimes also called parameters.

fix(x, v) fixes the variable *x* to the value *v*.

fix(x) fixes *x* to the value *x.value*, which might be the value obtained by solving another problem involving the variable *x*.

To allow the variable *x* to vary again, call *free!(x)*.

Fixing and freeing variables can be particularly useful as a tool for performing alternating minimization on nonconvex problems. For example, we can find an approximate solution to a nonnegative matrix factorization problem with alternating minimization as follows. We use warmstarts to speed up the solution.

```
# initialize nonconvex problem
n, k = 10, 1
A = rand(n, k) * rand(k, n)
x = Variable(n, k)
y = Variable(k, n)
problem = minimize(sum_squares(A - x*y), x>=0, y>=0)

# initialize value of y
y.value = rand(k, n)
# we'll do 10 iterations of alternating minimization
for i=1:10
        # first solve for x
        # with y fixed, the problem is convex
        fix!(y)
        solve!(problem, warmstart = i > 1 ? true : false)
        free!(y)

        # now solve for y with x fixed at the previous solution
        fix!(x)
        solve!(problem, warmstart = true)
        free!(x)
end
```

1.9 Credits

Currently, Convex.jl is developed and maintained by:

- Jenny Hong
- Karanveer Mohan
- Madeleine Udell
- David Zeng

The Convex.jl developers also thank:

- the JuliaOpt team: Iain Dunning, Joey Huchette and Miles Lubin
- Stephen Boyd, co-author of the book Convex Optimization
- Steven Diamond, developer of CVXPY and of a DCP tutorial website to teach disciplined convex programming.
- Michael Grant, developer of CVX.
- John Duchi and Hongseok Namkoong for developing the representation of power cones in terms of SOCP constraints used in this package.