CVXOPT User's Guide

Joachim Dahl & Lieven Vandenberghe

Release 1.0 – April 24, 2008

Copyright and License

The CVXOPT distribution includes source code for part of the SuiteSparse suite of sparse matrix algorithms, including:

- AMD Version 2.2.0. Copyright (c) 2007 by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff.
- CHOLMOD Version 1.6.0. Copyright (c) 2005-2007 by University of Florida, Timothy A. Davis and W. Hager.
- COLAMD version 2.7.0. Copyright (c) 1998-2007 by Timothy A. Davis.
- UMFPACK Version 5.2.0. Copyright (c) 1995-2006 by Timothy A. Davis.

These packages are licensed under the terms of the GNU General Public License, version 2 or higher[3] (UMFPACK, the Supernodal module of CHOLMOD) and the GNU Lesser General Public License, version 2.1 or higher[4] (the other CHOLMOD modules, AMD, COLAMD). For copyright and license details, consult the README files in the source directories or the website listed below.

Availability: www.cise.ufl.edu/research/sparse[5].

[1] http://www.gnu.org/licenses/gpl-3.0.html
[2] http://www.gnu.org/licenses/gpl-3.0.html
[3] http://www.gnu.org/licenses/old-licenses/gpl-2.0.html
[4] http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html
[5] http://www.cise.ufl.edu/research/sparse

Chapter 1

Introduction

CVXOPT is a free software package for convex optimization based on the Python programming language. It can be used with the interactive Python interpreter, on the command line by executing Python scripts, or integrated in other software via Python extension modules. Its main purpose is to make the development of software for convex optimization applications straightforward by building on Python's extensive standard library and on the strengths of Python as a high-level programming language.
Release 1.0 of CVXOPT includes routines for basic linear algebra calculations, interfaces to efficient libraries for solving dense and sparse linear equations, convex optimization solvers written in Python, interfaces to a few other optimization libraries, and a modeling tool for piecewise-linear convex optimization problems. These components are organized in different modules.

`cvxopt.base` This module defines a Python type `matrix` for storing and manipulating dense matrices, a Python type `spmatrix` for storing and manipulating sparse matrices, routines for generating sparse dense matrices (see section 2.7), and routines for sparse matrix-vector and matrix-matrix multiplication (see chapters 2 and 6).

`cvxopt.blas` Interface to most of the double-precision real and complex BLAS (chapter 3).

`cvxopt.lapack` Interface to the dense double-precision real and complex linear equation solvers and eigenvalue routines from LAPACK (chapter 4).

`cvxopt.fftw` An optional interface to the discrete transform routines from FFTW (section 5).

`cvxopt.amd` Interface to the approximate minimum degree ordering routine from AMD (chapter 7.1).

`cvxopt.umfpack` Interface to the sparse LU solver from UMFPACK (section 7.2).

`cvxopt.cholmod` Interface to the sparse Cholesky solver from CHOLMOD (section 7.3).

`cvxopt.solvers` Convex optimization routines and optional interfaces to solvers from GLPK, MOSEK and DSDP5 (chapters 8 and 9).

`cvxopt.modeling` Routines for specifying and solving linear programs and convex optimization problems with piecewise-linear cost and constraint functions (chapter 10).

`cvxopt.info` Defines a string `version` with the version number of the CVXOPT installation and a function `license()` that prints the CVXOPT license.

`cvxopt.printing` Contains functions and parameters that control how matrices are formatted.

The modules are described in detail in this manual and in the on-line Python help facility pydoc. Several example scripts are included in the distribution.

Chapter 2

Dense Matrices (cvxopt.base)

The cvxopt.base module defines two new Python types: matrix objects, used for dense matrix computations, and spmatrix objects, used for sparse matrix computations. In this chapter we describe the dense matrix object. Sparse matrices are discussed in chapter 6.

2.1 Creating Matrices

A matrix object is created by calling the function matrix(). The arguments specify the values of the coefficients, the dimensions, and the type (integer, double or complex) of the matrix.

matrix(x[, size[, tc]])

size is a tuple of length two with the matrix dimensions. The number of rows and/or the number of columns can be zero.

tc stands for typecode. The possible values are 'i', 'd' and 'z', for integer, real (double) and complex matrices, respectively.

x can be a number, a sequence of numbers, a dense or sparse matrix, a one- or two-dimensional NumPy array, or a list of lists of matrices and numbers.

- If x is a number (Python integer, float or complex), a matrix is created with the dimensions specified by size and with all the coefficients equal to x. The default value of size is $(1, 1)$, and the default value of tc is the type of x. If necessary, the type of x is converted (from integer to double when used to create a matrix of type 'd', and from integer or double to complex when used to create a matrix of type 'z').

```
>>> from cvxopt.base import matrix
>>> A = matrix(1, (1,4))
>>> print A
[ 1  1  1  1]
>>> A = matrix(1.0, (1,4))
>>> print A
[ 1.00e+00  1.00e+00  1.00e+00  1.00e+00]
>>> A = matrix(1+1j)
>>> print A
[ 1.00e+00+j1.00e+00]
```

- If x is a sequence of numbers (list, tuple, array array, xrange object, one-dimensional NumPy array, ...), then the numbers are interpreted as the coefficients of a matrix in column-major order. The length of x must be equal to the product of size[0] and size[1]. If size is not specified, a matrix with one

column is created. If `tc` is not specified, it is determined from the elements of `x` (and if that is impossible, for example because `x` is an empty list, a value `'i'` is used). Type conversion takes place as for scalar `x`. The following example shows several ways to define the same integer matrix.

```
>>> A = matrix([0, 1, 2, 3], (2,2))
>>> A = matrix((0, 1, 2, 3), (2,2))
>>> A = matrix(xrange(4), (2,2))
>>> from array import array
>>> A = matrix(array('i', [0,1,2,3]), (2,2))
>>> print A
[ 0  2]
[ 1  3]
```

- If `x` is a dense or sparse matrix (a `matrix` or a `spmatrix` object), or a two-dimensional NumPy array of type `'i'`, `'d'` or `'z'`, then the coefficients of `x` are copied, in column-major order, to a new matrix of the given size. The total number of elements in the new matrix (the product of `size[0]` and `size[1]`) must be the same as the product of the dimensions of `x`. If `size` is not specified, the dimensions of `x` are used. The default value of `tc` is the type of `x`. Type conversion takes place when the type of `x` differs from `tc`, in a similar way as for scalar `x`.

```
>>> A = matrix([1., 2., 3., 4., 5., 6.], (2,3))
>>> print A
[ 1.00e+00  3.00e+00  5.00e+00]
[ 2.00e+00  4.00e+00  6.00e+00]
>>> B = matrix(A, (3,2))
>>> print B
[ 1.00e+00  4.00e+00]
[ 2.00e+00  5.00e+00]
[ 3.00e+00  6.00e+00]
>>> C = matrix(B, tc='z')
>>> print C
[ 1.00e+00-j0.00e+00  4.00e+00-j0.00e+00]
[ 2.00e+00-j0.00e+00  5.00e+00-j0.00e+00]
[ 3.00e+00-j0.00e+00  6.00e+00-j0.00e+00]
>>> from numpy import array
>>> x = array([[1., 2., 3.], [4., 5., 6.]])
>>> x
array([[ 1.  2.  3.]
       [ 4.  5.  6.]])
>>> print matrix(x)
[ 1.00e+00  2.00e+00  3.00e+00]
[ 4.00e+00  5.00e+00  6.00e+00]
```

- If `x` is a list of lists of matrices (`matrix` or `spmatrix` objects) or numbers (Python `integer`, `float` or `complex`), then each element of `x` is interpreted as a block-column stored in column-major order. If `size` is not specified, the block-columns are juxtaposed to obtain a matrix with `len(x)` block-columns. If `size` is specified, then the matrix with `len(x)` block-columns is resized by copying its elements in column-major order into a matrix of the dimensions given by `size`. If `tc` is not specified, it is determined from the elements of `x` (and if that is impossible, for example because `x` is a list of empty lists, a value `'i'` is used). The same rules for type conversion apply as for scalar `x`.

```
>>> print matrix([[1., 2.], [3., 4.], [5., 6.]])
[ 1.00e+00  3.00e+00  5.00e+00]
[ 2.00e+00  4.00e+00  6.00e+00]
```

```
>>> A1 = matrix([1, 2], (2,1))
>>> B1 = matrix([6, 7, 8, 9, 10, 11], (2,3))
>>> B2 = matrix([12, 13, 14, 15, 16, 17], (2,3))
>>> B3 = matrix([18, 19, 20], (1,3))
>>> C = matrix([[A1, 3.0, 4.0, 5.0], [B1, B2, B3]])
>>> print C
[ 1.00e+00  6.00e+00  8.00e+00  1.00e+01]
[ 2.00e+00  7.00e+00  9.00e+00  1.10e+01]
[ 3.00e+00  1.20e+01  1.40e+01  1.60e+01]
[ 4.00e+00  1.30e+01  1.50e+01  1.70e+01]
[ 5.00e+00  1.80e+01  1.90e+01  2.00e+01]
```

A matrix with a single block-column can be represented by a single list (i.e., when the length of `x` is one, it can be replaced with `x[0]`).

```
>>> D = matrix([B1, B2, B3])
>>> print D
[  6   8  10]
[  7   9  11]
[ 12  14  16]
[ 13  15  17]
[ 18  19  20]
```

2.2 Attributes and Methods

A `matrix` has the following attributes.

size

A tuple with the dimensions of the matrix. The size of the matrix can be changed by altering this attribute, as long as the number of elements in the matrix remains unchanged.

typecode

A `char`, either `'i'`, `'d'`, or `'z'`, for integer, real and complex matrices, respectively. A read-only attribute.

trans()

Returns the transpose of the matrix as a new matrix. One can also use `A.T` instead of `A.trans()`.

ctrans()

Returns the conjugate transpose of the matrix as a new matrix. One can also use `A.H` instead of `A.ctrans()`.

real()

For complex matrices, returns the real part as a real matrix. For integer and real matrices, returns a copy of the matrix.

imag()

For complex matrices, returns the imaginary part as a real matrix. For integer and real matrices, returns an integer or real zero matrix.

__array_struct__

A PyCObject implementing the NumPy array interface (see section 2.8 for details).

tofile(`f`)

Writes the elements of the matrix in column-major order to a binary file `f`.

fromfile(`f`)

Reads the contents of a binary file `f` into the matrix object.

The last two methods are illustrated in the following example.

```
>>> from cvxopt.base import matrix
>>> A = matrix([[1.,2.,3.], [4.,5.,6.]])
>>> print A
[ 1.00e+00  4.00e+00]
[ 2.00e+00  5.00e+00]
[ 3.00e+00  6.00e+00]
>>> f = open('mat.bin','w')
>>> A.tofile(f)
>>> f.close()
>>> B = matrix(0.0, (2,3))
>>> f = open('mat.bin','r')
>>> B.fromfile(f)
>>> f.close()
>>> print B
[ 1.00e+00  3.00e+00  5.00e+00]
[ 2.00e+00  4.00e+00  6.00e+00]
```

Matrices can also be written to or read from files using the `dump()` and `load()` functions in the `pickle` module.

2.3 Arithmetic Operations

The following table lists the arithmetic operations defined for dense matrices. In this table `A` and `B` are dense matrices with compatible dimensions, `c` is a scalar (a Python number or a dense 1 by 1 matrix), and `d` is a Python number.

Unary plus/minus	`+A`, `-A`
Addition	`A+B`, `A+c`, `c+A`
Subtraction	`A-B`, `A-c`, `c-A`
Matrix multiplication	`A*B`
Scalar multiplication and division	`c*A`, `A*c`, `A/c`
Remainder after division	`A%c`
Elementwise exponentiation	`A**d`

If `c` in the expressions `A+c`, `c+A`, `A-c`, `c-A` is a number, then it is interpreted as a matrix with the same dimensions as `A`, type given by the type of `c`, and all entries equal to `c`. If `c` is a 1 by 1 matrix and `A` is not 1 by 1, then `c` is interpreted as a matrix with the same size of `A` and all entries equal to `c[0]`.
Postmultiplying a matrix with a number `c` means the same as premultiplying, i.e., scalar multiplication. Dividing a matrix by `c` means dividing all entries by `c`. If `c` is a 1 by 1 matrix and the product `c*A` or `A*c` cannot be interpreted as a matrix-matrix product, then it is interpreted as `c[0]*A`. The division `A/c` and remainder `A%c` with `c` a 1 by 1 matrix are always interpreted as `A/c[0]`, resp., `A%c[0]`.
If one of the operands in the arithmetic operations is integer (a scalar `integer` or a matrix of type `'i'`) and the other operand is double (a scalar `float` or a matrix of type `'d'`), then the integer operand is converted to double, and the result is a matrix of type `'d'`. If one of the operands is integer or double, and the other operand is complex (a scalar `complex` or a matrix of type `'z'`), then the first operand is converted to complex, and the result is a matrix of type `'z'`.

The result of `A**d` is a complex matrix if `A` or `d` are complex, and real otherwise.
Note that Python rounds the result of an integer division towards minus infinity.
The following in-place operations are also defined, but only if they do not change the type of the matrix `A`:

In-place addition	`A+=B`, `A+=c`
In-place subtraction	`A-=B`, `A-=c`
In-place scalar multiplication and division	`A*=c`, `A/=c`
In-place remainder	`A%=c`

For example, if `A` has type `'i'`, then `A+=B` is allowed if `B` has type `'i'`. It is not allowed if `B` has type `'d'` or `'z'` because the addition `A+B` results in a matrix of type `'d'` or `'z'` and therefore cannot be assigned to `A` without changing its type.
In-place matrix-matrix products are not allowed. (Except when `c` is a 1 by 1 matrix, in which case `A*=c` is interpreted as the scalar product `A*=c[0]`.)
It is important to know when a matrix operation creates a new object. The following rules apply.

- A simple assignment ("`A = B`") is given the standard Python interpretation, i.e., it assigns to the variable `A` a reference (or pointer) to the object referenced by `B`.

```
>>> B = matrix([[1.,2.], [3.,4.]])
>>> print B
[ 1.00e+00  3.00e+00]
[ 2.00e+00  4.00e+00]
>>> A = B
>>> A[0,0] = -1
>>> print B   # modifying A[0,0] also modified B[0,0]
[-1.00e+00  3.00e+00]
[ 2.00e+00  4.00e+00]
```

- The regular (i.e., not in-place) arithmetic operations always return new objects. Hence "`A = +B`" is equivalent to "`A = matrix(B)`".

```
>>> B = matrix([[1.,2.], [3.,4.]])
>>> A = +B
>>> A[0,0] = -1
>>> print B   # modifying A[0,0] does not modify B[0,0]
[ 1.00e+00  3.00e+00]
[ 2.00e+00  4.00e+00]
```

- The in-place operations directly modify the coefficients of the existing matrix object and do not create a new object.

```
>>> B = matrix([[1.,2.], [3.,4.]])
>>> A = B
>>> A *= 2
>>> print B   # in-place operation also changed B
[ 2.00e+00  6.00e+00]
[ 4.00e+00  8.00e+00]
>>> A = 2*A
>>> print B   # regular operation creates a new A, so does not change B
[ 2.00e+00  6.00e+00]
[ 4.00e+00  8.00e+00]
```

2.4 Indexing and Slicing

Matrices can be indexed using one or two arguments. In single-argument indexing of a matrix `A`, the index runs from `-len(A)` to `len(A)-1`, and is interpreted as an index in the one-dimensional array of coefficients of `A` in column-major order. Negative indices have the standard Python interpretation: for negative `k`, `A[k]` is the same element as `A[len(A)+k]`.
Four different types of one-argument indexing are implemented.

1. The index can be a single integer. This returns a number, e.g., `A[0]` is the first element of `A`.

2. The index can be an integer matrix. This returns a column matrix: the command "`A[matrix([0,1,2,3])]`" returns the 4 by 1 matrix consisting of the first four elements of `A`. The size of the index matrix is ignored: "`A[matrix([0,1,2,3], (2,2))]`" returns the same 4 by 1 matrix.

3. The index can be a list of integers. This returns a column matrix, e.g., `A[[0,1,2,3]]` is the 4 by 1 matrix consisting of elements 0, 1, 2, 3 of `A`.

4. The index can be a Python slice. This returns a matrix with one column (possibly 0 by 1, or 1 by 1). For example, `A[::2]` is the column matrix defined by taking every other element of `A`, stored in column-major order. `A[0:0]` is a matrix with size (0,1).

Thus, single-argument indexing returns a scalar (if the index is an integer), or a matrix with one column. This is consistent with the interpretation that single-argument indexing accesses the matrix in column-major order.
Note that an index list or an index matrix are equivalent, but they are both useful, especially when we perform operations on index sets. For example, if `I` and `J` are lists then `I+J` is the concatenated list, and `2*I` is `I` repeated twice. If they are matrices, these operations are interpreted as arithmetic operations. For large index sets, indexing with integer matrices is also faster than indexing with lists.
The following example illustrates one-argument indexing.

```
>>> from cvxopt.base import matrix
>>> A = matrix(range(16), (4,4), 'd')
>>> print A
[ 0.00e+00  4.00e+00  8.00e+00  1.20e+01]
[ 1.00e+00  5.00e+00  9.00e+00  1.30e+01]
[ 2.00e+00  6.00e+00  1.00e+01  1.40e+01]
[ 3.00e+00  7.00e+00  1.10e+01  1.50e+01]
>>> A[4]
4.0
>>> I = matrix([0, 5, 10, 15])
>>> print A[I]      # the diagonal
[ 0.00e+00]
[ 5.00e+00]
[ 1.00e+01]
[ 1.50e+01]
>>> I = [0,2];  J = [1,3]
>>> print A[2*I+J]  # duplicate I and append J
[ 0.00e+00]
[ 2.00e+00]
[ 0.00e+00]
[ 2.00e+00]
[ 1.00e+00]
[ 3.00e+00]
>>> I = matrix([0, 2]);  J =  matrix([1, 3])
>>> print A[2*I+J]  # multiply I by 2 and add J
```

```
[ 1.00e+00]
[ 7.00e+00]
>>> print A[4::4]   # get every fourth element skipping the first four
[ 4.00e+00]
[ 8.00e+00]
[ 1.20e+01]
```

In two-argument indexing the arguments can be any combinations of the four types listed above. The first argument indexes the rows of the matrix and the second argument indexes the columns. If both indices are scalars, then a scalar is returned. In all other cases, a matrix is returned. We continue the example.

```
>>> print A[:,1]
[ 4.00e+00]
[ 5.00e+00]
[ 6.00e+00]
[ 7.00e+00]
>>> J = matrix([0, 2])
>>> print A[J,J]
[ 0.00e+00  8.00e+00]
[ 2.00e+00  1.00e+01]
>>> print A[:2, -2:]
[ 8.00e+00  1.20e+01]
[ 9.00e+00  1.30e+01]
```

Expressions of the form `A[I]` or `A[I,J]` can also appear on the lefthand side of an assignment. The righthand side must be a scalar (i.e., a number or a 1 by 1 dense matrix), a sequence of numbers, or a dense or sparse matrix. If the righthand side is a scalar, it is interpreted as a matrix with identical entries and the dimensions of the lefthand side. If the righthand side is a sequence of numbers (list, tuple, `array` array, xrange object, ...) its values are interpreted as the coefficients of the lefthand side in column-major order. If the righthand side is a matrix (`matrix` or `spmatrix`), it must have the same size as the lefthand side. Sparse matrices are converted to dense in the assignment.
Indexed assignments are only allowed if they do not change the type of the matrix. For example, if `A` is a matrix with type `'d'`, then `A[I] = B` is only permitted if `B` is an `integer`, a `float`, or a matrix of type `'i'` or `'d'`. If `A` is an integer matrix, then `A[I] = B` is only permitted if `B` is an `integer` or an integer matrix.
The following example illlustrates indexed assignment.

```
>>> A = matrix(range(16), (4,4))
>>> A[::2,::2] = matrix([[-1, -2], [-3, -4]])
>>> print A
[ -1   4  -3  12]
[  1   5   9  13]
[ -2   6  -4  14]
[  3   7  11  15]
>>> A[::5] += 1
>>> print A
[  0   4  -3  12]
[  1   6   9  13]
[ -2   6  -3  14]
[  3   7  11  16]
>>> A[0,:] = -1, 1, -1, 1
>>> print A
[ -1   1  -1   1]
[  1   6   9  13]
[ -2   6  -3  14]
[  3   7  11  16]
```

```
>>> A[2:,2:] = xrange(4)
>>> print A
[ -1   1  -1   1]
[  1   6   9  13]
[ -2   6   0   2]
[  3   7   1   3]
```

2.5 Built-in Functions

Many Python built-in functions and operations can be used with matrix arguments. We list some useful examples.

len(x)

Returns the product of the number of rows and the number of columns.

bool([x])

Returns `False` if `x` is empty (i.e., `len(x)` is zero) and `True` otherwise.

max(x)

Returns the maximum element of `x`.

min(x)

Returns the minimum element of `x`.

abs(x)

Returns a matrix with the absolute values of the elements of `x`.

sum(x[, start=0.0])

Returns the sum of `start` and the elements of `x`.

Matrices can be used as arguments to the `list()`, `tuple()`, `zip()`, `map()`, and `filter()` functions described in section 2.1 of the Python Library Reference. `list(A)` and `tuple(A)` construct a list, respectively a tuple, from the elements of `A`. `zip(A,B,...)` returns a list of tuples, with the *i*th tuple containing the *i*th elements of `A`, `B`, ...

```
>>> from cvxopt.base import matrix
>>> A = matrix([[-11., -5., -20.], [-6., -0., 7.]])
>>> B = matrix(range(6), (3,2))
>>> list(A)
[-11.0, -5.0, -20.0, -6.0, 0.0, 7.0]
>>> tuple(B)
(0, 1, 2, 3, 4, 5)
>>> zip(A,B)
[(-11.0, 0), (-5.0, 1), (-20.0, 2), (-6.0, 3), (0.0, 4), (7.0, 5)]
```

`map(f,A)`, where `f` is a function and `A` is a matrix, returns a list constructed by applying `f` to each element of `A`. Multiple arguments can be provided, for example, as in `map(f,A,B)`, if `f` is a function with two arguments.

```
>>> A = matrix([[5, -4, 10, -7], [-1, -5, -6, 2], [6, 1, 5, 2],  [-1, 2, -3, -7]])
>>> B = matrix([[4,-15, 9, -14], [-4, -12, 1, -22], [-10, -9, 9, 12], [-9, -7,-11, -6]])
>>> print matrix(map(max, A, B), (4,4))   # takes componentwise maximum
[  5  -1   6  -1]
[ -4  -5   1   2]
[ 10   1   9  -3]
[ -7   2  12  -6]
```

`filter(f,A)`, where `f` is a function and `A` is a matrix, returns a list containing the elements of `A` for which `f` is true.

```
>>> filter(lambda x: x%2, A)          # list of odd elements in A
[5, -7, -1, -5, 1, 5, -1, -3, -7]
>>> filter(lambda x: -2 < x < 3, A)   # list of elements between -2 and 3
[-1, 2, 1, 2, -1, 2]
```

It is also possible to iterate over matrix elements, as illustrated in the following example.

```
>>> A = matrix([[5, -3], [9, 11]])
>>> for x in A: print max(x,0)
...
5
0
9
11
>>> [max(x,0) for x in A]
[5, 0, 9, 11]
```

The expression "`x in A`" returns `True` if an element of `A` is equal to `x` and `False` otherwise.

2.6 Other Matrix Functions

The following functions of dense matrices can be imported from `cvxopt.base`.

sqrt(x)

The elementwise square root of `x`. The result is returned as a real matrix if `x` is an integer or real matrix and as a complex matrix if `x` is a complex matrix. Raises an exception when `x` is an integer or real matrix with negative elements.

sin(x)

The sine function applied elementwise to `x`. The result is returned as a real matrix if `x` is an integer or real matrix and as a complex matrix otherwise.

cos(x)

The cosine function applied elementwise to `x`. The result is returned as a real matrix if `x` is an integer or real matrix and as a complex matrix otherwise.

exp(x)

The exponential function applied elementwise to `x`. The result is returned as a real matrix if `x` is an integer or real matrix and as a complex matrix otherwise.

log(x)

The natural logarithm applied elementwise to `x`. The result is returned as a real matrix if `x` is an integer or real matrix and as a complex matrix otherwise. Raises an exception when `x` is an integer or real matrix with nonnegative elements, or a complex matrix with zero elements.

mul(x, y)

The elementwise product of `x` and `y`. The two matrices must have the same size and type.

div(x, y)

The elementwise division of `x` by `y`. The two matrices must have the same size and type.

2.7 Randomly Generated Matrices

The `cvxopt.base` module provides two functions `normal()` and `uniform()` for generating randomly distributed matrices. The default installation relies on the pseudo-random number generators in the Python standard library `random`. Alternatively, the random number generators in the GNU Scientific Library (GSL)[1] can be used, if this option is selected during the installation of CVXOPT. The random matrix functions based on GSL are faster than the default functions based on the `random` module.

normal(`nrows[, ncols[, mean[, std]]]`)

Returns a type `'d'` matrix of size `nrows` by `ncols` with elements chosen from a normal distribution with mean `mean` and standard deviation `std`. The default values for the optional arguments are `ncols`=1, `mean`=0.0, `std`=1.0.

uniform(`nrows[, ncols[, a[, b]]]`)

Returns a type `'d'` matrix of size `nrows` by `ncols` matrix with elements uniformly distributed between `a` and `b`. The default values for the optional arguments are `ncols`=1, `a`=0.0, `b`=1.0.

setseed([`value`])

Sets the state of the random number generator. `value` must be an integer. If `value` is absent or equal to zero, the value is taken from the system clock. If the Python random number generators are used, this is equivalent to `random.seed(value)`.

getseed()

Returns the current state of the random number generator. This function is only available if the GSL random number generators are installed. (The state of the random number generators in the Python `random` module can be managed via the functions `random.getstate()` and `random.setstate()`.)

2.8 The NumPy Array Interface

The CVXOPT `matrix` object is compatible with the NumPy Array Interface, which allows Python objects that represent multidimensional arrays to exchange data using information stored in the attribute `__array_struct__`.

See also:

- NumPy Array Interface Specification[2]
- NumPy home page[3]

As already mentioned in section 2.1, a two-dimensional array object (for example, a NumPy matrix or two-dimensional array) can be converted to a CVXOPT `matrix` object by using the `matrix()` constructor. Conversely, CVXOPT matrices can be used as array-like objects in NumPy. The following example illustrates the compatibility of CVXOPT matrices and NumPy arrays.

```
>>> from cvxopt.base import matrix
>>> a = matrix(range(6), (2,3), 'd')
>>> print a
[ 0.00e+00  2.00e+00  4.00e+00]
[ 1.00e+00  3.00e+00  5.00e+00]
>>> from numpy import array
>>> b = array(a)
```

[1] `http://www.gnu.org/software/gsl`
[2] `http://numpy.scipy.org/array_interface.shtml`
[3] `http://numpy.scipy.org`

```
>>> b
array([[ 0.  2.  4.]
       [ 1.  3.  5.]])
>>> a*b
array([[  0.   4.  16.]
       [  1.   9.  25.]])
>>> from numpy import mat
>>> c = mat(a)
>>> c
matrix([[ 0.  2.  4.]
        [ 1.  3.  5.]])
>>> a.T * c
matrix([[  1.,   3.,   5.],
        [  3.,  13.,  23.],
        [  5.,  23.,  41.]])
```

In the first product, `a*b` is interpreted as NumPy array multiplication, i.e., componentwise multiplication. The second product `a.T*c` is interpreted as NumPy matrix multiplication, i.e., standard matrix multiplication.

Chapter 3

The BLAS Interface (`cvxopt.blas`)

The `cvxopt.blas` module provides an interface to the double-precision real and complex Basic Linear Algebra Subprograms (BLAS). The names and calling sequences of the Python functions in the interface closely match the corresponding Fortran BLAS routines (described in the references below) and their functionality is exactly the same. Many of the operations performed by the BLAS routines can be implemented in a more straightforward way by using the matrix arithmetic of section 2.3, combined with the slicing and indexing of section 2.4. As an example, "`C = A*B`" gives the same result as the BLAS call "`gemm(A,B,C)`". The BLAS interface offers two advantages. First, some of the functions it includes are not easily implemented using the basic matrix arithmetic. For example, BLAS includes functions that efficiently exploit symmetry or triangular matrix structure. Second, there is a performance difference that can be significant for large matrices. Although our implementation of the basic matrix arithmetic makes internal calls to BLAS, it also often requires creating temporary matrices to store intermediate results. The BLAS functions on the other hand always operate directly on their matrix arguments and never require any copying to temporary matrices. Thus they can be viewed as generalizations of the in-place matrix addition and scalar multiplication of section 2.3 to more complicated operations.

See also:

- C. L. Lawson, R. J. Hanson, D. R. Kincaid, F. T. Krogh, Basic Linear Algebra Subprograms for Fortran Use, ACM Transactions on Mathematical Software, 5(3), 309-323, 1975.
- J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, An Extended Set of Fortran Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, 14(1), 1-17, 1988.
- J. J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, A Set of Level 3 Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, 16(1), 1-17, 1990.

3.1 Matrix Classes

The BLAS exploit several types of matrix structure: symmetric, Hermitian, triangular, and banded. We represent all these matrix classes by dense real or complex `matrix` objects, with additional arguments that specify the structure.

Vector A real or complex n-vector is represented by a `matrix` of type `'d'` or `'z'` and length n, with the entries of the vector stored in column-major order.

General matrix A general real or complex m by n matrix is represented by a real or complex `matrix` of size (m, n).

Symmetric matrix A real or complex symmetric matrix of order n is represented by a real or complex `matrix` of size (n, n), and a character argument `uplo` with two possible values: `'L'` and `'U'`. If `uplo` is `'L'`, the lower

triangular part of the symmetric matrix is stored; if `uplo` is `'U'`, the upper triangular part is stored. A square `matrix X` of size (n, n) can therefore be used to represent the symmetric matrices

$$\begin{bmatrix} X[0,0] & X[1,0] & X[2,0] & \cdots & X[n-1,0] \\ X[1,0] & X[1,1] & X[2,1] & \cdots & X[n-1,1] \\ X[2,0] & X[2,1] & X[2,2] & \cdots & X[n-1,2] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X[n-1,0] & X[n-1,1] & X[n-1,2] & \cdots & X[n-1,n-1] \end{bmatrix} \quad \text{if uplo = 'L',}$$

$$\begin{bmatrix} X[0,0] & X[0,1] & X[0,2] & \cdots & X[0,n-1] \\ X[0,1] & X[1,1] & X[1,2] & \cdots & X[1,n-1] \\ X[0,2] & X[1,2] & X[2,2] & \cdots & X[2,n-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X[0,n-1] & X[1,n-1] & X[2,n-1] & \cdots & X[n-1,n-1] \end{bmatrix} \quad \text{if uplo = U'.}$$

Complex Hermitian matrix A complex Hermitian matrix of order n is represented by a `matrix` of type `'z'` and size (n, n), and a character argument `uplo` with the same meaning as for symmetric matrices. A complex `matrix X` of size (n, n) can represent the 8 Hermitian matrices

$$\begin{bmatrix} \Re X[0,0] & \bar{X}[1,0] & \bar{X}[2,0] & \cdots & \bar{X}[n-1,0] \\ X[1,0] & \Re X[1,1] & \bar{X}[2,1] & \cdots & \bar{X}[n-1,1] \\ X[2,0] & X[2,1] & \Re X[2,2] & \cdots & \bar{X}[n-1,2] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X[n-1,0] & X[n-1,1] & X[n-1,2] & \cdots & \Re X[n-1,n-1] \end{bmatrix} \quad \text{if uplo = 'L',}$$

$$\begin{bmatrix} \Re X[0,0] & X[0,1] & X[0,2] & \cdots & X[0,n-1] \\ \bar{X}[0,1] & \Re X[1,1] & X[1,2] & \cdots & X[1,n-1] \\ \bar{X}[0,2] & \bar{X}[1,2] & \Re X[2,2] & \cdots & X[2,n-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \bar{X}[0,n-1] & \bar{X}[1,n-1] & \bar{X}[2,n-1] & \cdots & \Re X[n-1,n-1] \end{bmatrix} \quad \text{if uplo = 'U'.}$$

Triangular matrix A real or complex triangular matrix of order n is represented by a real or complex `matrix` of size (n, n), and two character arguments: an argument `uplo` with possible values `'L'` and `'U'` to distinguish between lower and upper triangular matrices, and an argument `diag` with possible values `'U'` and `'N'` to distinguish between unit and non-unit triangular matrices. A square `matrix X` of size (n, n) can represent the triangular matrices

$$\begin{bmatrix} X[0,0] & 0 & 0 & \cdots & 0 \\ X[1,0] & X[1,1] & 0 & \cdots & 0 \\ X[2,0] & X[2,1] & X[2,2] & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X[n-1,0] & X[n-1,1] & X[n-1,2] & \cdots & X[n-1,n-1] \end{bmatrix} \quad \text{if uplo = 'L' and diag = 'N',}$$

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ X[1,0] & 1 & 0 & \cdots & 0 \\ X[2,0] & X[2,1] & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X[n-1,0] & X[n-1,1] & X[n-1,2] & \cdots & 1 \end{bmatrix} \quad \text{if uplo = 'L' and diag = 'U',}$$

$$\begin{bmatrix} X[0,0] & X[0,1] & X[0,2] & \cdots & X[0,n-1] \\ 0 & X[1,1] & X[1,2] & \cdots & X[1,n-1] \\ 0 & 0 & X[2,2] & \cdots & X[2,n-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & X[n-1,n-1] \end{bmatrix}$$ if uplo = 'U' and diag = 'N',

$$\begin{bmatrix} 1 & X[0,1] & X[0,2] & \cdots & X[0,n-1] \\ 0 & 1 & X[1,2] & \cdots & X[1,n-1] \\ 0 & 0 & 1 & \cdots & X[2,n-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$ if uplo = 'U' and diag = 'U'.

General band matrix A general real or complex m by n band matrix with k_l subdiagonals and k_u superdiagonals is represented by a real or complex `matrix X` of size ($k_l + k_u + 1$, n), and the two integers m and k_l. The diagonals of the band matrix are stored in the rows of `X`, starting at the top diagonal, and shifted horizontally so that the entries of the kth column of the band matrix are stored in column k of `X`. A `matrix X` of size ($k_l + k_u + 1$, n) therefore represents the m by n band matrix

$$\begin{bmatrix} X[k_u,0] & X[k_u-1,1] & X[k_u-2,2] & \cdots & X[0,k_u] & 0 & \cdots \\ X[k_u+1,0] & X[k_u,1] & X[k_u-1,2] & \cdots & X[1,k_u] & X[0,k_u+1] & \cdots \\ X[k_u+2,0] & X[k_u+1,1] & X[k_u,2] & \cdots & X[2,k_u] & X[1,k_u+1] & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ X[k_u+k_l,0] & X[k_u+k_l-1,1] & X[k_u+k_l-2,2] & \cdots & & & \\ 0 & X[k_u+k_l,1] & X[k_u+k_l-1,2] & \cdots & & & \\ \vdots & \vdots & \vdots & \ddots & & & \end{bmatrix}.$$

Symmetric band matrix A real or complex symmetric band matrix of order n with k subdiagonals, is represented by a real or complex matrix `X` of size ($k + 1$, n), and an argument *uplo* to indicate whether the subdiagonals (*uplo* is `'L'`) or superdiagonals (*uplo* is `'U'`) are stored. The $k + 1$ diagonals are stored as rows of `X`, starting at the top diagonal (i.e., the main diagonal if *uplo* is `'L'`, or the kth superdiagonal if *uplo* is `'U'`) and shifted horizontally so that the entries of the kth column of the band matrix are stored in column k of `X`. A `matrix X` of size ($k + 1$, n) can therefore represent the band matrices

$$\begin{bmatrix} X[0,0] & X[1,0] & X[2,0] & \cdots & X[k,0] & 0 & \cdots \\ X[1,0] & X[0,1] & X[1,1] & \cdots & X[k-1,1] & X[k,1] & \cdots \\ X[2,0] & X[1,1] & X[0,2] & \cdots & X[k-2,2] & X[k-1,2] & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ X[k,0] & X[k-1,1] & X[k-2,2] & \cdots & & & \\ 0 & X[k,1] & X[k-1,2] & \cdots & & & \\ \vdots & \vdots & \vdots & \ddots & & & \end{bmatrix}$$ if uplo = 'L',

$$\begin{bmatrix} X[k,0] & X[k-1,1] & X[k-2,2] & \cdots & X[0,k] & 0 & \cdots \\ X[k-1,1] & X[k,1] & X[k-1,2] & \cdots & X[1,k] & X[0,k+1] & \cdots \\ X[k-2,2] & X[k-1,2] & X[k,2] & \cdots & X[2,k] & X[1,k+1] & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ X[0,k] & X[1,k] & X[2,k] & \cdots & & & \\ 0 & X[0,k+1] & X[1,k+1] & \cdots & & & \\ \vdots & \vdots & \vdots & \ddots & & & \end{bmatrix}$$ if uplo='U'.

Hermitian band matrix A complex Hermitian band matrix of order n with k subdiagonals is represented by a complex matrix of size ($k + 1$, n) and an argument `uplo`. A `matrix X` of size ($k + 1$, n) can represent the band

matrices

$$\begin{bmatrix} \Re X[0,0] & \bar{X}[1,0] & \bar{X}[2,0] & \cdots & \bar{X}[k,0] & 0 & \cdots \\ X[1,0] & \Re X[0,1] & \bar{X}[1,1] & \cdots & \bar{X}[k-1,1] & \bar{X}[k,1] & \cdots \\ X[2,0] & X[1,1] & \Re X[0,2] & \cdots & \bar{X}[k-2,2] & \bar{X}[k-1,2] & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ X[k,0] & X[k-1,1] & X[k-2,2] & \cdots & & & \\ 0 & X[k,1] & X[k-1,2] & \cdots & & & \\ \vdots & \vdots & \vdots & \ddots & & & \end{bmatrix} \quad \text{if uplo = 'L',}$$

$$\begin{bmatrix} \Re X[k,0] & X[k-1,1] & X[k-2,2] & \cdots & X[0,k] & 0 & \cdots \\ \bar{X}[k-1,1] & \Re X[k,1] & X[k-1,2] & \cdots & X[1,k] & X[0,k+1] & \cdots \\ \bar{X}[k-2,2] & \bar{X}[k-1,2] & \Re X[k,2] & \cdots & X[2,k] & X[1,k+1] & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ \bar{X}[0,k] & \bar{X}[1,k] & \bar{X}[2,k] & \cdots & & & \\ 0 & \bar{X}[0,k+1] & \bar{X}[1,k+1] & \cdots & & & \\ \vdots & \vdots & \vdots & \ddots & & & \end{bmatrix} \quad \text{if uplo='U'.}$$

Triangular band matrix A triangular band matrix of order n with k subdiagonals or superdiagonals is represented by a real complex matrix of size $(k+1, n)$ and two character arguments `uplo` and `diag`. A `matrix X` of size $(k+1, n)$ can represent the band matrices

$$\begin{bmatrix} X[0,0] & 0 & 0 & \cdots \\ X[1,0] & X[0,1] & 0 & \cdots \\ X[2,0] & X[1,1] & X[0,2] & \cdots \\ \vdots & \vdots & \vdots & \ddots \\ X[k,0] & X[k-1,1] & X[k-2,2] & \cdots \\ 0 & X[k,1] & X[k-1,1] & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad \text{if uplo = 'L' and diag = 'N',}$$

$$\begin{bmatrix} 1 & 0 & 0 & \cdots \\ X[1,0] & 1 & 0 & \cdots \\ X[2,0] & X[1,1] & 1 & \cdots \\ \vdots & \vdots & \vdots & \ddots \\ X[k,0] & X[k-1,1] & X[k-2,2] & \cdots \\ 0 & X[k,1] & X[k-1,2] & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad \text{if uplo = 'L' and diag = 'U',}$$

$$\begin{bmatrix} X[k,0] & X[k-1,1] & X[k-2,3] & \cdots & X[0,k] & 0 & \cdots \\ 0 & X[k,1] & X[k-1,2] & \cdots & X[1,k] & X[0,k+1] & \cdots \\ 0 & 0 & X[k,2] & \cdots & X[2,k] & X[1,k+1] & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \end{bmatrix} \quad \text{if uplo = 'U' and diag = 'N',}$$

$$\begin{bmatrix} 1 & X[k-1,1] & X[k-2,3] & \cdots & X[0,k] & 0 & \cdots \\ 0 & 1 & X[k-1,2] & \cdots & X[1,k] & X[0,k+1] & \cdots \\ 0 & 0 & 1 & \cdots & X[2,k] & X[1,k+1] & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \end{bmatrix} \quad \text{if uplo = 'U' and diag = 'U'.}$$

When discussing BLAS functions in the following sections we will omit several less important optional arguments that can be used to select submatrices for in-place operations. The complete specification is documented in the docstrings of the source code and the pydoc help program.

3.2 Level 1 BLAS

The level 1 functions implement vector operations.

scal(`alpha, x`)

Scales a vector by a constant:

$$x := \alpha x.$$

If `x` is a real `matrix`, the scalar argument `alpha` must be a Python `integer` or `float`. If `x` is complex, `alpha` can be an `integer`, `float`, or `complex`.

nrm2(`x`)

Euclidean norm of a vector: returns

$$\|x\|_2.$$

asum(`x`)

1-Norm of a vector: returns

$$\|x\|_1 \quad (x \text{ real}), \qquad \|\Re x\|_1 + \|\Im x\|_1 \quad (x \text{ complex}).$$

iamax(`x`)

Returns

$$\operatorname*{argmax}_{k=0,\ldots,n-1} |x_k| \quad (x \text{ real}), \qquad \operatorname*{argmax}_{k=0,\ldots,n-1} |\Re x_k| + |\Im x_k| \quad (x \text{ complex}).$$

If more than one coefficient achieves the maximum, the index of the first k is returned.

swap(`x, y`)

Interchanges two vectors:

$$x \leftrightarrow y.$$

`x` and `y` are matrices of the same type (`'d'` or `'z'`).

copy(`x, y`)

Copies a vector to another vector:

$$y := x.$$

`x` and `y` are matrices of the same type (`'d'` or `'z'`).

axpy(`x, y[,alpha=1.0]`)

Constant times a vector plus a vector:

$$y := \alpha x + y.$$

`x` and `y` are matrices of the same type (`'d'` or `'z'`). If `x` is real, the scalar argument `alpha` must be a Python `integer` or `float`. If `x` is complex, `alpha` can be an `integer`, `float`, or `complex`.

dot(`x, y`)

Returns

$$x^H y.$$

`x` and `y` are matrices of the same type (`'d'` or `'z'`).

dotu(`x, y`)

Returns

$$x^T y.$$

`x` and `y` are matrices of the same type (`'d'` or `'z'`).

3.3 Level 2 BLAS

The level 2 functions implement matrix-vector products and rank-1 and rank-2 matrix updates. Different types of matrix structure can be exploited using the conventions of section 3.1.

gemv(`A, x, y[, trans='N'[, alpha=1.0[, beta=0.0]]]`)

Matrix-vector product with a general matrix:

$$y := \alpha Ax+\beta y \quad (\text{trans} = \text{'N'}), \qquad y := \alpha A^T x+\beta y \quad (\text{trans} = \text{'T'}), \qquad y := \alpha A^H x+\beta y \quad (\text{trans} = \text{'C'}).$$

The arguments `A`, `x` and `y` must have the same type (`'d'` or `'z'`). Complex values of `alpha` and `beta` are only allowed if `A` is complex.

symv(`A, x, y[, uplo='L'[, alpha=1.0[, beta=0.0]]]`)

Matrix-vector product with a real symmetric matrix:

$$y := \alpha Ax + \beta y,$$

where A is a real symmetric matrix. The arguments `A`, `x` and `y` must have type `'d'` and `alpha` and `beta` must be real.

hemv(`A, x, y[, uplo='L'[, alpha=1.0[, beta=0.0]]]`)

Matrix-vector product with a real symmetric or complex Hermitian matrix:

$$y := \alpha Ax + \beta y,$$

where A is real symmetric or complex Hermitian. The arguments `A`, `x` and `y` must have the same type (`'d'` or `'z'`). Complex values of `alpha` and `beta` are only allowed if `A` is complex.

trmv(`A, x[, uplo='L'[, trans='N'[, diag='N']]]`)

Matrix-vector product with a triangular matrix:

$$x := Ax \quad (\text{trans} = \text{'N'}), \qquad x := A^T x \quad (\text{trans} = \text{'T'}), \qquad x := A^H x \quad (\text{trans} = \text{'C'}),$$

where A is square and triangular. The arguments `A` and `x` must have the same type (`'d'` or `'z'`).

trsv(`A, x[, uplo='L'[, trans='N'[, diag='N']]]`)

Solution of a nonsingular triangular set of linear equations:

$$x := A^{-1}x \quad (\text{trans} = \text{'N'}), \qquad x := A^{-T}x \quad (\text{trans} = \text{'T'}), \qquad x := A^{-H}x \quad (\text{trans} = \text{'C'}),$$

where A is square and triangular with nonzero diagonal elements. The arguments `A` and `x` must have the same type (`'d'` or `'z'`).

gbmv(`A, m, kl, x, y[, trans='N' [, alpha=1.0[, beta=0.0]]]`)

Matrix-vector product with a general band matrix:

$$y := \alpha Ax+\beta y \quad (\text{trans} = \text{'N'}), \qquad y := \alpha A^T x+\beta y \quad (\text{trans} = \text{'T'}), \qquad y := \alpha A^H x+\beta y \quad (\text{trans} = \text{'C'}),$$

where A is a rectangular band matrix with m rows and k_l subdiagonals. The arguments `A`, `x` and `y` must have the same type (`'d'` or `'z'`). Complex values of `alpha` and `beta` are only allowed if `A` is complex.

sbmv(`A, x, y[, uplo='L'[, alpha=1.0[, beta=0.0]]]`)

Matrix-vector product with a real symmetric band matrix:

$$y := \alpha Ax + \beta y,$$

where A is a real symmetric band matrix. The arguments `A`, `x` and `y` must have type `'d'` and `alpha` and `beta` must be real.

hbmv(`A, x, y[, uplo='L'[, alpha=1.0[, beta=0.0]]]`)

Matrix-vector product with a real symmetric or complex Hermitian band matrix:

$$y := \alpha Ax + \beta y,$$

where A is a real symmetric or complex Hermitian band matrix. The arguments `A`, `x` and `y` must have the same type (`'d'` or `'z'`). Complex values of `alpha` and `beta` are only allowed if `A` is complex.

tbmv(`A, x[, uplo='L'[, trans[, diag]]]`)

Matrix-vector product with a triangular band matrix:

$$x := Ax \quad (\text{trans} = \text{'N'}), \qquad x := A^T x \quad (\text{trans} = \text{'T'}), \qquad x := A^H x \quad (\text{trans} = \text{'C'}).$$

The arguments `A` and `x` must have the same type (`'d'` or `'z'`).

tbsv(`A, x[, uplo='L'[, trans[, diag]]]`)

Solution of a triangular banded set of linear equations:

$$x := A^{-1}x \quad (\text{trans} = \text{'N'}), \qquad x := A^{-T}x \quad (\text{trans} = \text{'T'}), \qquad x := A^{-H}x \quad (\text{trans} = \text{'T'}),$$

where A is a triangular band matrix of with nonzero diagonal elements. The arguments `A` and `x` must have the same type (`'d'` or `'z'`).

ger(`x, y, A[, alpha=1.0]`)

General rank-1 update:

$$A := A + \alpha xy^H,$$

where A is a general matrix. The arguments `A`, `x` and `y` must have the same type (`'d'` or `'z'`). Complex values of `alpha` are only allowed if `A` is complex.

geru(`x, y, A[, alpha=1.0]`)

General rank-1 update:

$$A := A + \alpha xy^T,$$

where A is a general matrix. The arguments `A`, `x` and `y` must have the same type (`'d'` or `'z'`). Complex values of `alpha` are only allowed if `A` is complex.

syr(`x, A[, uplo='L'[, alpha=1.0]]`)

Symmetric rank-1 update:

$$A := A + \alpha xx^T,$$

where A is a real symmetric matrix. The arguments `A` and `x` must have type `'d'`. `alpha` must be a real number.

her(`x, A[, uplo='L'[, alpha=1.0]]`)

Hermitian rank-1 update:

$$A := A + \alpha xx^H,$$

where A is a real symmetric or complex Hermitian matrix. The arguments `A` and `x` must have the same type (`'d'` or `'z'`). `alpha` must be a real number.

syr2(`x, y, A[, uplo='L'[, alpha=1.0]]`)

Symmetric rank-2 update:

$$A := A + \alpha(xy^T + yx^T),$$

where A is a real symmetric matrix. The arguments `A`, `x` and `y` must have type `'d'`. `alpha` must be real.

her2(`x, y, A[, uplo='L'[, alpha=1.0]]`)

Symmetric rank-2 update:

$$A := A + \alpha xy^H + \bar{\alpha} yx^H,$$

where A is a a real symmetric or complex Hermitian matrix. The arguments `A`, `x` and `y` must have the same type (`'d'` or `'z'`). Complex values of `alpha` are only allowed if `A` is complex.

As an example, the following code multiplies the tridiagonal matrix

$$A = \begin{bmatrix} 1 & 6 & 0 & 0 \\ 2 & -4 & 3 & 0 \\ 0 & -3 & -1 & 1 \end{bmatrix}$$

with the vector *x* = (1,-1,2,-2).

```
>>> from cvxopt.base import matrix
>>> from cvxopt.blas import gbmv
>>> A = matrix([[0., 1., 2.],  [6., -4., -3.],  [3., -1., 0.],  [1., 0., 0.]])
>>> x = matrix([1., -1., 2., -2.])
>>> y = matrix(0., (3,1))
>>> gbmv(A, 3, 1, x, y)
>>> print y
[-5.00e+00]
[ 1.20e+01]
[-1.00e+00]
```

The following example illustrates the use of `tbsv()`.

```
>>> from cvxopt.base import matrix
>>> from cvxopt.blas import tbsv
>>> A = matrix([-6., 5., -1., 2.], (1,4))
>>> x = matrix(1.0, (4,1))
>>> tbsv(A, x)  # x := diag(A)^{-1}*x
>>> print x
[-1.67e-01]
[ 2.00e-01]
[-1.00e+00]
[ 5.00e-01]
```

3.4 Level 3 BLAS

The level 3 BLAS include functions for matrix-matrix multiplication.
gemm(`A, B, C[, transA='N'[, transB='N'[, alpha=1.0[, beta=0.0]]]]`)

Matrix-matrix product of two general matrices:

$$C := \alpha\,\mathrm{op}(A)\,\mathrm{op}(B) + \beta C$$

where

$$\mathrm{op}(A) = \begin{cases} A & \mathrm{transA} = \mathrm{'N'} \\ A^T & \mathrm{transA} = \mathrm{'T'} \\ A^H & \mathrm{transA} = \mathrm{'C'} \end{cases} \qquad \mathrm{op}(B) = \begin{cases} B & \mathrm{transB} = \mathrm{'N'} \\ B^T & \mathrm{transB} = \mathrm{'T'} \\ B^H & \mathrm{transB} = \mathrm{'C'}. \end{cases}$$

The arguments `A`, `B` and `C` must have the same type (`'d'` or `'z'`). Complex values of `alpha` and `beta` are only allowed if `A` is complex.

symm(`A, B, C[, side='L'[, uplo='L'[, alpha=1.0[, beta=0.0]]]]`)

Product of a real or complex symmetric matrix A and a general matrix B:

$$C := \alpha AB + \beta C \quad (\mathrm{side} = \mathrm{'L'}), \qquad C := \alpha BA + \beta C \quad (\mathrm{side} = \mathrm{'R'}).$$

The arguments `A`, `B` and `C` must have the same type (`'d'` or `'z'`). Complex values of `alpha` and `beta` are only allowed if `A` is complex.

hemm(`A, B, C[, side='L'[, uplo='L'[, alpha=1.0[, beta=0.0]]]]`)

Product of a real symmetric or complex Hermitian matrix *A* and a general matrix *B*:

$$C := \alpha AB + \beta C \quad (\mathrm{side} = \mathrm{'L'}), \qquad C := \alpha BA + \beta C \quad (\mathrm{side} = \mathrm{'R'}).$$

The arguments `A`, `B` and `C` must have the same type (`'d'` or `'z'`). Complex values of `alpha` and `beta` are only allowed if `A` is complex.

trmm(`A, B[, side='L'[, uplo='L'[, transA='N'[, diag='N'[, alpha=1.0]]]]]`)

Product of a triangular matrix *A* and a general matrix *B*:

$$B := \alpha\,\mathrm{op}(A)B \quad (\mathrm{side} = \mathrm{'L'}), \qquad B := \alpha B\,\mathrm{op}(A) \quad (\mathrm{side} = \mathrm{'R'}), \qquad \mathrm{op}(A) = \begin{cases} A & \mathrm{transA} = \mathrm{'N'} \\ A^T & \mathrm{transA} = \mathrm{'T'} \\ A^H & \mathrm{transA} = \mathrm{'C'}. \end{cases}$$

The arguments `A` and `B` must have the same type (`'d'` or `'z'`). Complex values of `alpha` are only allowed if `A` is complex.

trsm(`A, B[, side='L'[, uplo='L'[, transA='N'[, diag='N'[, alpha=1.0]]]]]`)

Solution of a nonsingular triangular system of equations:

$$B := \alpha\,\mathrm{op}(A)^{-1}B \quad (\mathrm{side} = \mathrm{'L'}), \qquad B := \alpha B\,\mathrm{op}(A)^{-1} \quad (\mathrm{side} = \mathrm{'R'}), \qquad \mathrm{op}(A) = \begin{cases} A & \mathrm{transA} = \mathrm{'N'} \\ A^T & \mathrm{transA} = \mathrm{'T'} \\ A^H & \mathrm{transA} = \mathrm{'C'}, \end{cases}$$

where A is triangular and B is a general matrix. The arguments `A` and `B` must have the same type (`'d'` or `'z'`). Complex values of `alpha` are only allowed if `A` is complex.

syrk(`A, C[, uplo='L'[, trans='N'[, alpha=1.0[, beta=0.0]]]]`)

Rank-*k* update of a real or complex symmetric matrix *C*:

$$C := \alpha AA^T + \beta C \quad (\mathrm{trans} = \mathrm{'N'}), \qquad C := \alpha A^TA + \beta C \quad (\mathrm{trans} = \mathrm{'T'}),$$

where A is a general matrix. The arguments `A` and `C` must have the same type (`'d'` or `'z'`). Complex values of `alpha` and `beta` are only allowed if `A` is complex.

herk(`A, C[, uplo='L'[, trans='N'[, alpha=1.0[, beta=0.0]]]]`)

Rank-k update of a real symmetric or complex Hermitian matrix C:

$$C := \alpha AA^H + \beta C \quad (\text{trans} = \text{'N'}), \qquad C := \alpha A^H A + \beta C \quad (\text{trans} = \text{'C'}),$$

where A is a general matrix. The arguments `A` and `C` must have the same type (`'d'` or `'z'`). `alpha` and `beta` must be real.

syr2k(`A, B, C[, uplo='L'[, trans='N'[, alpha=1.0[, beta=0.0]]]]`)

Rank-*2k* update of a real or complex symmetric matrix *C*:

$$C := \alpha(AB^T + BA^T) + \beta C \quad (\text{trans} = \text{'N'}), \qquad C := \alpha(A^T B + B^T A) + \beta C \quad (\text{trans} = \text{'T'}).$$

A and B are general real or complex matrices. The arguments `A`, `B` and `C` must have the same type. Complex values of `alpha` and `beta` are only allowed if `A` is complex.

her2k(`A, B, C[, uplo='L'[, trans='N'[, alpha=1.0[ beta=0.0]]]]`)

Rank-$2k$ update of a real symmetric or complex Hermitian matrix C:

$$C := \alpha AB^H + \bar{\alpha} BA^H + \beta C \quad (\text{trans} = \text{'N'}), \qquad C := \alpha A^H B + \bar{\alpha} B^H A + \beta C \quad (\text{trans} = \text{'C'}),$$

where A and B are general matrices. The arguments `A`, `B` and `C` must have the same type (`'d'` or `'z'`). Complex values of `alpha` are only allowed if `A` is complex. `beta` must be real.

Chapter 4

The LAPACK Interface (`cvxopt.lapack`)

The module `cvxopt.lapack` includes functions for solving dense sets of linear equations, for the corresponding matrix factorizations (LU, Cholesky, $\mathrm{LDL}^{\mathrm{T}}$), for solving least-squares and least-norm problems, for QR factorization, for symmetric eigenvalue problems and for singular value decomposition.
In this chapter we briefly describe the Python calling sequences. For further details on the underlying LAPACK functions we refer to the LAPACK Users' Guide and manual pages.
The BLAS conventional storage scheme of section 3.1 is used. As in the previous chapter, we omit from the function definitions less important arguments that are useful for selecting submatrices. The complete definitions are documented in the docstrings in the source code.
See also: LAPACK Users' Guide, Third Edition, SIAM, 1999[1].

4.1 General Linear Equations

gesv(`A, B[, ipiv=None]`)

Solves

$$AX = B,$$

where A and B are real or complex matrices, with A square and nonsingular. On exit, `B` is replaced by the solution. The arguments `A` and `B` must have the same type (`'d'` or `'z'`). The optional argument `ipiv` is an integer matrix of length at least n. If `ipiv` is provided, then `gesv()` solves the system, replaces `A` with its triangular factors, and returns the permutation matrix in `ipiv`. If `ipiv` is not specified, then `gesv()` solves the system but does not return the LU factorization and does not modify `A`. For example,

```
>>> gesv(A, B)
```

solves the system without modifying `A` and returns the solution in `B`.

```
>>> gesv(A, B, ipiv)
```

returns the solution in `B` and also returns the details of the LU factorization in `A` and `ipiv`.

Raises an `ArithmeticError` if the matrix is singular.

getrf(`A, ipiv`)

[1] `http://www.netlib.org/lapack/lug/lapack_lug.html`

LU factorization of a general, possibly rectangular, real or complex matrix,

$$A = PLU$$

where A is m by n. The argument `ipiv` is an integer matrix of length at least $\min\{m, n\}$. On exit, the lower triangular part of `A` is replaced by L, the upper triangular part by U, and the permutation matrix is returned in `ipiv`. Raises an `ArithmeticError` if the matrix is not full rank.

getrs(`A, ipiv, B[, trans='N']`)

Solves a general set of linear equations

$$AX = B \quad (\text{trans} = \text{'N'}), \qquad A^T X = B \quad (\text{trans} = \text{'T'}), \qquad A^H X = B \quad (\text{trans} = \text{'C'}),$$

given the LU factorization computed by `gesv()` or `getrf()`. On entry, `A` and `ipiv` must contain the factorization as computed by `gesv()` or `getrf()`. On exit, `B` is overwritten with the solution. `B` must have the same type as `A`.

getri(`A, ipiv`)

Computes the inverse of a matrix. On entry, `A` and `ipiv` must contain the factorization as computed by `gesv()` or `getrf()`. On exit, `A` contains the inverse.

In the following example we compute

$$x = (A^{-1} + A^{-T})b$$

for randomly generated problem data, factoring the coefficient matrix once.

```
>>> from cvxopt.base import matrix, normal
>>> from cvxopt.lapack import gesv, getrs
>>> n = 10
>>> A = normal(n,n)
>>> b = normal(n)
>>> ipiv = matrix(0, (n,1))
>>> x = +b
>>> gesv(A, x, ipiv)                # x = A^{-1}*b
>>> x2 = +b
>>> getrs(A, ipiv, x2, trans='T')   # x2 = A^{-T}*b
>>> x += x2
```

Separate functions are provided for equations with band matrices.

gbsv(`A, kl, B[, ipiv=None]`)

Solves

$$AX = B,$$

where A and B are real or complex matrices, with A n by n and banded with k_l subdiagonals. The arguments `A` and `B` must have the same type (`'d'` or `'z'`).

The optional argument `ipiv` is an integer matrix of length at least n. If `ipiv` is provided, then `A` must have $2k_l + k_u + 1$ rows. On entry the diagonals of A are stored in rows $k_l + 1$ to $2k_l + k_u + 1$ of the `A`, using the BLAS format for general band matrices (see section 3.1). On exit, the factorization is returned in `A` and `ipiv`.

If `ipiv` is not provided, then `A` must have $k_l + k_u + 1$ rows. On entry the diagonals of A are stored in the rows of `A`, following the standard format for general band matrices. In this case, `gbsv()` does not modify `A` on exit and does not return the factorization.

On exit, `B` is replaced by the solution X. Raises an `ArithmeticError` if the matrix is singular.

gbtrf(`A, m, kl, ipiv`)

LU factorization of a general m by n real or complex band matrix with k_l subdiagonals. The matrix is stored using the BLAS format for general band matrices (see section 3.1), by providing the diagonals (stored as rows of a $k_u + k_l + 1$ by n matrix), the number of rows m, and the number of subdiagonals k_l. The argument `ipiv` is an integer matrix of length at least $\min\{m, n\}$. On exit, `A` and `ipiv` contain the details of the factorization. Raises an `ArithmeticError` if the matrix is not full rank.

gbtrs(`A, kl, ipiv, B[, trans='N']`)

Solves a set of linear equations

$$AX = B \quad (\text{trans} = \text{'N'}), \qquad A^T X = B \quad (\text{trans} = \text{'T'}), \qquad A^H X = B \quad (\text{trans} = \text{'C'}),$$

with A a general band matrix with k_l subdiagonals, given the LU factorization computed by `gbsv()` or `gbtrf()`. On entry, `A` and `ipiv` must contain the factorization as computed by `gbsv()` or `gbtrf()`. On exit, `B` is overwritten with the solution. `B` must have the same type as `A`.

As an example, we solve a linear equation with

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 6 & 7 & 8 & 9 \\ 0 & 10 & 11 & 12 \end{bmatrix}, \qquad x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

```
>>> from cvxopt.base import matrix
>>> from cvxopt.lapack import gbsv, gbtrf, gbtrs
>>> n, kl, ku = 4, 2, 1
>>> A = matrix([[0., 1., 3., 6.], [2., 4., 7., 10.], [5., 8., 11., 0.], [9., 12., 0., 0.]])
>>> x = matrix(1.0, (4,1))
>>> gbsv(A, kl, x)
>>> print x
[ 7.14e-02]
[ 4.64e-01]
[-2.14e-01]
[-1.07e-01]
```

The code below illustrates how one can reuse the factorization returned by `gbsv()`.

```
>>> Ac = matrix(0.0, (2*kl+ku+1,n))
>>> Ac[kl:,:] = A
>>> ipiv = matrix(0, (n,1))
>>> x = matrix(1.0, (4,1))
>>> gbsv(Ac, kl, x, ipiv)                 # solves A*x = 1
>>> print x
[ 7.14e-02]
[ 4.64e-01]
[-2.14e-01]
[-1.07e-01]
>>> x = matrix(1.0, (4,1))
>>> gbtrs(Ac, kl, ipiv, x, trans='T')     # solve A^T*x = 1
>>> print x
[ 7.14e-02]
[ 2.38e-02]
[ 1.43e-01]
[-2.38e-02]
```

An alternative method uses `gbtrf()` for the factorization.

```
>>> Ac[kl:,:] = A
>>> gbtrf(Ac, n, kl, ipiv)
>>> x = matrix(1.0, (4,1))
>>> gbtrs(Ac, kl, ipiv, x)                 # solve A^T*x = 1
>>> print x
[ 7.14e-02]
[ 4.64e-01]
[-2.14e-01]
[-1.07e-01]
>>> x = matrix(1.0, (4,1))
>>> gbtrs(Ac, kl, ipiv, x, trans='T')      # solve A^T*x = 1
>>> print x
[ 7.14e-02]
[ 2.38e-02]
[ 1.43e-01]
[-2.38e-02]
```

The following functions can be used for tridiagonal matrices. They use a simpler matrix format, that stores the diagonals in three separate vectors.

gtsv(`dl, d, du, B)`)

Solves

$$AX = B,$$

where A is an n by n tridiagonal matrix, with subdiagonal stored as a matrix `dl` of length $n-1$, diagonal stored as a matrix `d` of length n, and superdiagonal stored as a matrix `du` of length $n-1$. The four arguments must have the same type (`'d'` or `'z'`). On exit `dl`, `d`, `du` are overwritten with the details of the LU factorization of A, and `B` is overwritten with the solution X. Raises an `ArithmeticError` if the matrix is singular.

gttrf(`dl, d, du, du2, ipiv`)

LU factorization of an n by n tridiagonal matrix with subdiagonal d_l, diagonal d and superdiagonal d_u. `dl`, `d` and `du` must have the same type. `du2` is a matrix of length $n-2$, and of the same type as `dl`. `ipiv` is an `'i'` matrix of length n. On exit, the five arguments contain the details of the factorization. Raises an `ArithmeticError` if the matrix is singular.

pttrs(`dl, d, du, du2, ipiv, B[, trans='N']`)

Solves a set of linear equations

$$AX = B \quad (\text{trans} = \text{'N'}), \qquad A^T X = B \quad (\text{trans} = \text{'T'}), \qquad A^H X = B \quad (\text{trans} = \text{'C'}),$$

where A is an n by n tridiagonal matrix. The arguments `dl`, `d`, `du`, `du2` and `ipiv` contain the details of the LU factorization as returned by `gttrf()`. On exit, `B` is overwritten with the solution X. `B` must have the same type as `dl`.

4.2 Positive Definite Linear Equations

posv(`A, B[, uplo='L']`)

Solves

$$AX = B,$$

where A is a real symmetric or complex Hermitian positive definite matrix. On exit, `B` is replaced by the solution, and `A` is overwritten with the Cholesky factor. The matrices `A` and `B` must have the same type (`'d'` or `'z'`). Raises an `ArithmeticError` if the matrix is not positive definite.

potrf(A[, uplo='L'])

Cholesky factorization

$$A = LL^T \qquad \text{or} \qquad A = LL^H$$

of a positive definite real symmetric or complex Hermitian matrix A. On exit, the lower triangular part of A (if uplo is 'L') or the upper triangular part (if uplo is 'U') is overwritten with the Cholesky factor or its (conjugate) transpose. Raises an ArithmeticError if the matrix is not positive definite.

potrs(A, B[, uplo='L'])

Solves a set of linear equations

$$AX = B$$

with a positive definite real symmetric or complex Hermitian matrix, given the Cholesky factorization computed by posv() or potrf(). On entry, A contains the triangular factor, as computed by posv() or potrf(). On exit, B is replaced by the solution. B must have the same type as A.

potri(A[, uplo='L'])

Computes the inverse of a positive definite matrix. On entry, A contains the Cholesky factorization computed by potrf() or posv(). On exit, it contains the inverse.

As an example, we use posv() to solve the linear system

$$\begin{bmatrix} -\mathbf{diag}\,(d)^2 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4.1}$$

by block-elimination. We first pick a random problem.

```
>>> from cvxopt.base import matrix, div, normal, uniform
>>> from cvxopt.blas import syrk, gemv
>>> from cvxopt.lapack import posv
>>> m, n = 100, 50
>>> A = normal(m,n)
>>> b1, b2 = normal(m), normal(n)
>>> d = uniform(m)
```

We then solve the equations

$$A^T\mathbf{diag}\,(d)^{-2}Ax_2 = b_2 + A^T\mathbf{diag}\,(d)^{-2}b_1, \qquad \mathbf{diag}\,(d)^2x_1 = Ax_2 - b_1.$$

```
>>> Asc = div(A, d[:, n*[0]])                 # Asc := diag(d)^{-1}*A
>>> B = matrix(0.0, (n,n))
>>> syrk(Asc, B, trans='T')                   # B := Asc^T * Asc = A^T * diag(d)^{-2} * A
>>> x1 = div(b1, d)                           # x1 := diag(d)^{-1}*b1
>>> x2 = +b2
>>> gemv(Asc, x1, x2, trans='T', beta=1.0)    # x2 := x2 + Asc^T*x1 = b2 + A^T*diag(d)^{-2}
>>> posv(B, x2)                               # x2 := B^{-1}*x2 = B^{-1}*(b2 + A^T*diag(d)^
>>> gemv(Asc, x2, x1, beta=-1.0)              # x1 := Asc*x2 - x1 = diag(d)^{-1} * (A*x2 - b
>>> x1 = div(x1, d)                           # x1 := diag(d)^{-1}*x1 = diag(d)^{-2} * (A*x2
```

There are separate routines for equations with positive definite band matrices.
pbsv(A, B[, uplo='L'])

Solves

$$AX = B$$

where A is a real symmetric or complex Hermitian positive definite band matrix. On entry, the diagonals of A are stored in `A`, using the BLAS format for symmetric or Hermitian band matrices (see section 3.1). On exit, `B` is replaced by the solution, and `A` is overwritten with the Cholesky factor (in the BLAS format for triangular band matrices). The matrices `A` and `B` must have the same type (`'d'` or `'z'`). Raises an `ArithmeticError` if the matrix is not positive definite.

pbtrf(`A[, uplo='L']`)

Cholesky factorization

$$A = LL^T \qquad \text{or} \qquad A = LL^H$$

of a positive definite real symmetric or complex Hermitian band matrix A. On entry, the diagonals of A are stored in `A`, using the BLAS format for symmetric or Hermitian band matrices. On exit, `A` contains the Cholesky factor, in the BLAS format for triangular band matrices. Raises an `ArithmeticError` if the matrix is not positive definite.

pbtrs(`A, B[, uplo='L']`)

Solves a set of linear equations

$$AX = B$$

with a positive definite real symmetric or complex Hermitian band matrix, given the Cholesky factorization computed by `pbsv()` or `pbtrf()`. On entry, `A` contains the triangular factor, as computed by `pbsv()` or `pbtrf()`. On exit, `B` is replaced by the solution. `B` must have the same type as `A`.

The following functions are useful for tridiagonal systems.
ptsv(`d, e, B`)

Solves

$$AX = B,$$

where A is an n by n positive definite real symmetric or complex Hermitian tridiagonal matrix. Its diagonal is stored as a `'d'` matrix `d` of length n and its subdiagonal as a `'d'` or `'z'` matrix `e` of length $n - 1$. The arguments `e` and `B` must have the same type. On exit `d` contains the diagonal elements of D in the $\mathrm{LDL}^{\mathrm{T}}$ or $\mathrm{LDL}^{\mathrm{H}}$ factorization of A, and `e` contains the subdiagonal elements of the unit lower bidiagonal matrix L. `B` is overwritten with the solution X. Raises an `ArithmeticError` if the matrix is singular.

pttrf(`d, e`)

$\mathrm{LDL}^{\mathrm{T}}$ or $\mathrm{LDL}^{\mathrm{H}}$ factorization of an n by n positive definite real symmetric or complex Hermitian tridiagonal matrix A. On entry, the argument `d` is a `'d'` matrix with the diagonal elements of A. The argument `e` is `'d'` or `'z'` matrix with the subdiagonal elements of A. On exit `d` contains the diagonal elements of D, and `e` contains the subdiagonal elements of the unit lower bidiagonal matrix L. Raises an `ArithmeticError` if the matrix is singular.

gttrs(`d, e, B[, uplo='L']`)

Solves a set of linear equations

$$AX = B$$

where A is an n by n positive definite real symmetric or complex Hermitian tridiagonal matrix, given its $\mathrm{LDL}^{\mathrm{T}}$ or $\mathrm{LDL}^{\mathrm{H}}$ factorization. The argument `d` is the diagonal of the diagonal matrix D. The argument `uplo` only matters for complex matrices. If `uplo` is `'L'`, then on exit `e` contains the subdiagonal elements of the unit bidiagonal matrix L. If `uplo` is `'U'`, then `e` contains the complex conjugates of the elements of the unit bidiagonal matrix L. On exit, `B` is overwritten with the solution X. `B` must have the same type as `e`.

4.3 Symmetric and Hermitian Linear Equations

sysv(`A, B[, ipiv=None[, uplo='L']]`)

Solves

$$AX = B$$

where A is a real or complex symmetric matrix of order n. On exit, `B` is replaced by the solution. The matrices `A` and `B` must have the same type (`'d'` or `'z'`). The optional argument `ipiv` is an integer matrix of length at least equal to n. If `ipiv` is provided, `sysv()` solves the system and returns the factorization in `A` and `ipiv`. If `ipiv` is not specified, `sysv()` solves the system but does not return the factorization and does not modify `A`. Raises an `ArithmeticError` if the matrix is singular.

sytrf(`A, ipiv[, uplo='L']`)

$\mathrm{LDL}^{\mathrm{T}}$ factorization

$$PAP^T = LDL^T$$

of a real or complex symmetric matrix A of order n. `ipiv` is an `'i'` matrix of length at least n. On exit, `A` and `ipiv` contain the factorization. Raises an `ArithmeticError` if the matrix is singular.

sytrs(`A, ipiv, B[, uplo='L']`)

Solves

$$AX = B$$

given the $\mathrm{LDL}^{\mathrm{T}}$ factorization computed by `sytrf()` or `sysv()`. `B` must have the same type as `A`.

sytri(`A, ipiv[, uplo='L']`)

Computes the inverse of a real or complex symmetric matrix. On entry, `A` and `ipiv` contain the $\mathrm{LDL}^{\mathrm{T}}$ factorization computed by `sytrf()` or `sysv()`. On exit, `A` contains the inverse.

hesv(`A, B[, ipiv=None[, uplo='L']]`)

Solves

$$AX = B$$

where A is a real symmetric or complex Hermitian of order n. On exit, `B` is replaced by the solution. The matrices `A` and `B` must have the same type (`'d'` or `'z'`). The optional argument `ipiv` is an integer matrix of length at least n. If `ipiv` is provided, then `hesv()` solves the system and returns the factorization in `A` and `ipiv`. If `ipiv` is not specified, then `hesv()` solves the system but does not return the factorization and does not modify `A`. Raises an `ArithmeticError` if the matrix is singular.

hetrf(`A, ipiv[, uplo='L']`)

$\mathrm{LDL}^{\mathrm{H}}$ factorization

$$PAP^T = LDL^H$$

of a real symmetric or complex Hermitian matrix of order n. `ipiv` is an `'i'` matrix of length at least n. On exit, `A` and `ipiv` contain the factorization. Raises an `ArithmeticError` if the matrix is singular.

hetrs(`A, ipiv, B[, uplo='L']`)

Solves

$$AX = B$$

given the $\mathrm{LDL}^{\mathrm{H}}$ factorization computed by `hetrf()` or `hesv()`.

hetri(`A, ipiv[, uplo='L']`)

Computes the inverse of a real symmetric or complex Hermitian matrix. On entry, `A` and `ipiv` contain the $\mathrm{LDL}^{\mathrm{H}}$ factorization computed by `hetrf()` or `hesv()`. On exit, `A` contains the inverse.

As an example we solve the KKT system (4.1).

```
>>> from cvxopt.lapack import sysv
>>> K = matrix(0.0, (m+n,m+n))
>>> K[: (m+n)*m : m+n+1] = -d**2
>>> K[:m, m:] = A
>>> x = matrix(0.0, (m+n,1))
>>> x[:m], x[m:] = b1, b2
>>> sysv(K, x, uplo='U')
```

4.4 Triangular Linear Equations

trtrs(`A, B[, uplo='L'[, trans='N'[, diag='N']]]`)

Solves a triangular set of equations

$$AX = B \quad (\text{trans} = \text{'N'}), \qquad A^T X = B \quad (\text{trans} = \text{'T'}), \qquad A^H X = B \quad (\text{trans} = \text{'C'}),$$

where A is real or complex and triangular of order n, and `B` is a matrix with n rows. `A` and `B` are matrices with the same type (`'d'` or `'z'`). `trtrs()` is similar to `blas.trsm()`, except that it raises an `ArithmeticError` if a diagonal element of `A` is zero (whereas `blas.trsm()` returns `inf` values).

trtri(`A[, uplo='L'[, diag='N']]`)

Computes the inverse of a real or complex triangular matrix A. On exit, `A` contains the inverse.

tbtrs(`A, B[, uplo='L'[, trans='T'[,diag='N']]]`)

Solves a triangular set of equations

$$AX = B \quad (\text{trans} = \text{'N'}), \qquad A^T X = B \quad (\text{trans} = \text{'T'}), \qquad A^H X = B \quad (\text{trans} = \text{'C'}),$$

where A is real or complex triangular band matrix of order n, and `B` is a matrix with n rows. The diagonals of A are stored in `A` using the BLAS conventions for triangular band matrices. `A` and `B` are matrices with the same type (`'d'` or `'z'`). On exit, `B` is replaced by the solution X.

4.5 Least-Squares and Least-Norm Problems

gels(`A, B[, trans='N']`)

Solves least-squares and least-norm problems with a full rank m by n matrix A.

1. `trans` is `'N'`. If m is greater than or equal to n, `gels()` solves the least-squares problem

$$\text{minimize} \quad \|AX - B\|_F.$$

If m is less than or equal to n, `gels()` solves the least-norm problem

$$\begin{array}{ll} \text{minimize} & \|X\|_F \\ \text{subject to} & AX = B. \end{array}$$

2. `trans` is `'T'` or `'C'` and `A` and `B` are real. If m is greater than or equal to n, `gels()` solves the least-norm problem

$$\begin{array}{ll} \text{minimize} & \|X\|_F \\ \text{subject to} & A^T X = B. \end{array}$$

If m is less than or equal to n, `gels()` solves the least-squares problem

$$\text{minimize} \quad \|A^T X - B\|_F.$$

3. `trans` is `'C'` and `A` and `B` are complex. If m is greater than or equal to n, `gels()` solves the least-norm problem

$$\begin{array}{ll} \text{minimize} & \|X\|_F \\ \text{subject to} & A^H X = B. \end{array}$$

If m is less than or equal to n, `gels()` solves the least-squares problem

$$\text{minimize} \quad \|A^H X - B\|_F.$$

`A` and `B` must have the same typecode (`'d'` or `'z'`). `trans = 'T'` is not allowed if `A` is complex. On exit, the solution X is stored as the leading submatrix of `B`. The array `A` is overwritten with details of the QR or the LQ factorization of A. Note that `gels()` does not check whether A is full rank.

geqrf(`A, tau`)

QR factorization of a real or complex matrix `A`:

$$A = QR.$$

If `A` is m by n, then Q is m by m and orthogonal/unitary, and `R` is m by n and upper triangular (if m is greater than or equal to n), or upper trapezoidal (if m is less than or equal to n). `tau` is a matrix of the same type as `A` and of length at least $\min\{m, n\}$. On exit, R is stored in the upper triangular part of `A`. The matrix Q is stored as a product of $\min\{m, n\}$ elementary reflectors in the first $\min\{m, n\}$ columns of `A` and in `tau`.

ormqr(`A, tau, C[, side='L'[, trans='N']]`)

Product with a real orthogonal matrix:

$$C := \mathrm{op}(Q)C \quad (\mathrm{side} = \mathrm{'L'}), \qquad C := C\,\mathrm{op}(Q) \quad (\mathrm{side} = \mathrm{'R'}), \qquad \mathrm{op}(Q) = \begin{cases} Q & \mathrm{trans} = \mathrm{'N'} \\ Q^T & \mathrm{trans} = \mathrm{'T'}, \end{cases}$$

where Q is square and orthogonal. Q is stored in `A` and `tau` as a product of min{`A.size`[0], `A.size`[1]} elementary reflectors, as computed by `geqrf()`.

unmqr(`A, tau, C[, side='L'[, trans='N']]`)

Product with a real orthogonal or complex unitary matrix:

$$C := \mathrm{op}(Q)C \quad (\mathrm{side} = \mathrm{'L'}), \qquad C := C\,\mathrm{op}(Q) \quad (\mathrm{side} = \mathrm{'R'}), \qquad \mathrm{op}(Q) = \begin{cases} Q & \mathrm{trans} = \mathrm{'N'} \\ Q^T & \mathrm{trans} = \mathrm{'T'} \\ Q^H & \mathrm{trans} = \mathrm{'C'}, \end{cases}$$

Q is square and orthogonal or unitary. Q is stored in `A` and `tau` as a product of min{`A.size`[0], `A.size`[1]} elementary reflectors, as computed by `geqrf()`. The arrays `A`, `tau` and `C` must have the same type. `trans = 'T'` is only allowed if the typecode is `'d'`.

In the following example, we solve a least-squares problem by a direct call to `gels()`, and by separate calls to `geqrf()`, `ormqr()`, and `trtrs()`.

```
>>> from cvxopt import random, blas, lapack
>>> from cvxopt.base import matrix
>>> m, n = 10, 5
>>> A, b = random.normal(m,n), random.normal(m,1)
>>> x1 = +b
>>> lapack.gels(+A, x1)                    # x1[:n] minimizes ||A*x1[:n] - b||_2
>>> tau = matrix(0.0, (n,1))
>>> lapack.geqrf(A, tau)                   # A = [Q1, Q2] * [R1; 0]
>>> x2 = +b
>>> lapack.ormqr(A, tau, x2, trans='T')  # x2 := [Q1, Q2]' * b
>>> lapack.trtrs(A[:n,:], x2, uplo='U')  # x2[:n] := R1^{-1}*x2[:n]
>>> blas.nrm2(x1[:n] - x2[:n])
3.0050798580569307e-16
```

4.6 Symmetric and Hermitian Eigenvalue Decomposition

The first four routines compute all or selected eigenvalues and eigenvectors of a real symmetric matrix A:

$$A = V\,\mathbf{diag}\,(\lambda)V^T, \qquad V^TV = I.$$

syev(`A, W[, jobz='N'[, uplo='L']]`)

Eigenvalue decomposition of a real symmetric matrix of order n. `W` is a real matrix of length at least n. On exit, `W` contains the eigenvalues in ascending order. If `jobz` is `'V'`, the eigenvectors are also computed and returned in `A`. If `jobz` is `'N'`, the eigenvectors are not returned and the contents of `A` are destroyed. Raises an `ArithmeticError` if the eigenvalue decomposition fails.

syevd(`A, W[, jobz='N'[, uplo='L']]`)

This is an alternative to `syev()`, based on a different algorithm. It is faster on large problems, but also uses more memory.

syevx(`A, W[, jobz='N'[, range='A'[, uplo='L'[, vl=0.0, vu=0.0[, il=1, iu=1[, Z=None]]]]]]`)

Computes selected eigenvalues and eigenvectors of a real symmetric matrix `A` of order n.

`W` is a real matrix of length at least n. On exit, `W` contains the eigenvalues in ascending order. If `range` is `'A'`, all the eigenvalues are computed. If `range` is `'I'`, eigenvalues i_l through i_u are computed, where $1 \leq i_l \leq i_u \leq n$. If `range` is `'V'`, the eigenvalues in the interval $(v_l, v_u]$ are computed.

If `jobz` is `'V'`, the (normalized) eigenvectors are computed, and returned in `Z`. If `jobz` is `'N'`, the eigenvectors are not computed. In both cases, the contents of `A` are destroyed on exit. `Z` is optional (and not referenced) if `jobz` is `'N'`. It is required if `jobz` is `'V'` and must have at least n columns if `range` is `'A'` or `'V'` and at least $i_u - i_l + 1$ columns if `range` is `'I'`.

`syevx()` returns the number of computed eigenvalues.

syevr(`A, W[, jobz='N'[, range='A'[, uplo='L'[, vl=0.0, vu=0.0[, il=1, iu=n[, Z=None]]]]]]`)

This is an alternative to `syevx()`. `syevr()` is the most recent LAPACK routine for symmetric eigenvalue problems, and expected to supersede the three other routines in future releases.

The next four routines can be used to compute eigenvalues and eigenvectors for complex Hermitian matrices:

$$A = V\,\mathbf{diag}\,(\lambda)V^H, \qquad V^HV = I.$$

For real symmetric matrices they are identical to the corresponding `syev_()` routines.

heev(`A, W[, jobz='N'[, uplo='L']]`)

Eigenvalue decomposition of a real symmetric or complex Hermitian matrix of order n. The calling sequence is identical to `syev()`, except that `A` can be real or complex.

heevd(`A, W[, jobz='N'[, uplo='L']]`)

This is an alternative to `heev()`.

heevx(`A, W[, jobz='N'[, range='A'[, uplo='L'[, vl=0.0, vu=0.0 [, il=1, iu=n[, Z=None]]]]]]`)

Computes selected eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix of order n. The calling sequence is identical to `syevx()`, except that `A` can be real or complex. `Z` must have the same type as `A`.

heevr(`A, W[, jobz='N'[, range='A'[, uplo='L'[, vl=0.0, vu=0.0[, il=1, iu=n[, Z=None]]]]]]`)

This is an alternative to `heevx()`.

4.7 Generalized Symmetric Definite Eigenproblems

Three types of generalized eigenvalue problems can be solved:

$$AZ = BZ\,\mathbf{diag}\,(\lambda) \quad \text{(type 1)}, \qquad ABZ = Z\,\mathbf{diag}\,(\lambda) \quad \text{(type 2)}, \qquad BAZ = Z\,\mathbf{diag}\,(\lambda) \quad \text{(type 3)}, \tag{4.2}$$

with A and B real symmetric or complex Hermitian, and B positive definite. The matrix of eigenvectors is normalized as follows:

$$Z^H BZ = I \quad \text{(types 1 and 2)}, \qquad Z^H B^{-1} Z = I \quad \text{(type 3)}.$$

sygv(`A, B, W[, itype=1[, jobz='N'[, uplo='L']]]`)

Solves the generalized eigenproblem (4.2) for real symmetric matrices of order n, stored in real matrices `A` and `B`. `itype` is an integer with possible values 1, 2, 3, and specifies the type of eigenproblem. `W` is a real matrix of length at least n. On exit, it contains the eigenvalues in ascending order. On exit, `B` contains the Cholesky factor of B. If `jobz` is `'V'`, the eigenvectors are computed and returned in `A`. If `jobz` is `'N'`, the eigenvectors are not returned and the contents of `A` are destroyed.

hegv(`A, B, W[, itype=1[, jobz='N'[, uplo='L']]]`)

Generalized eigenvalue problem (4.2) of real symmetric or complex Hermitian matrix of order n. The calling sequence is identical to `sygv()`, except that `A` and `B` can be real or complex.

4.8 Singular Value Decomposition

gesvd(`A, S[, jobu='N'[, jobvt='N'[, U=None[, Vt=None]]]]`)

Singular value decomposition

$$A = U\Sigma V^T, \qquad A = U\Sigma V^H$$

of a real or complex m by n matrix `A`.

`S` is a real matrix of length at least $\min\{m, n\}$. On exit, its first $\min\{m, n\}$ elements are the singular values in descending order.

The argument `jobu` controls how many left singular vectors are computed. The possible values are `'N'`, `'A'`, `'S'` and `'O'`. If `jobu` is `'N'`, no left singular vectors are computed. If `jobu` is `'A'`, all left singular vectors are computed and returned as columns of `U`. If `jobu` is `'S'`, the first $\min\{m, n\}$ left singular vectors are computed and returned as columns of `U`. If `jobu` is `'O'`, the first $\min\{m, n\}$ left singular vectors are computed

and returned as columns of `A`. The argument `U` is `None` (if `jobu` is `'N'` or `'A'`) or a matrix of the same type as `A`.

The argument `jobvt` controls how many right singular vectors are computed. The possible values are `'N'`, `'A'`, `'S'` and `'O'`. If `jobvt` is `'N'`, no right singular vectors are computed. If `jobvt` is `'A'`, all right singular vectors are computed and returned as rows of `Vt`. If `jobvt` is `'S'`, the first $\min\{m,n\}$ right singular vectors are computed and their (conjugate) transposes are returned as rows of `Vt`. If `jobvt` is `'O'`, the first $\min\{m,n\}$ right singular vectors are computed and their (conjugate) transposes are returned as rows of `A`. Note that the (conjugate) transposes of the right singular vectors (i.e., the matrix V^H) are returned in `Vt` or `A`. The argument `Vt` can be `None` (if `jobvt` is `'N'` or `'A'`) or a matrix of the same type as `A`.

On exit, the contents of `A` are destroyed.

gesdd(`A, S[, jobz='N'[, U=None[, Vt=None]]]`)

Singular value decomposition of a real or complex m by n matrix `A`. This function is based on a divide-and-conquer algorithm and is faster than `gesvd()`.

`S` is a real matrix of length at least $\min\{m,n\}$. On exit, its first $\min\{m,n\}$ elements are the singular values in descending order.

The argument `jobz` controls how many singular vectors are computed. The possible values are `'N'`, `'A'`, `'S'` and `'O'`. If `jobz` is `'N'`, no singular vectors are computed. If `jobz` is `'A'`, all m left singular vectors are computed and returned as columns of `U` and all n right singular vectors are computed and returned as rows of `Vt`. If `jobz` is `'S'`, the first $\min\{m,n\}$ left and right singular vectors are computed and returned as columns of `U` and rows of `Vt`. If `jobz` is `'O'` and m is greater than or equal to n, the first n left singular vectors are returned as columns of `A` and the n right singular vectors are returned as rows of `Vt`. If `jobz` is `'O'` and m is less than n, the m left singular vectors are returned as columns of `U` and the first m right singular vectors are returned as rows of `A`. Note that the (conjugate) transposes of the right singular vectors are returned in `Vt` or `A`.

The argument `U` can be `None` (if `jobz` is `'N'` or `'A'` of `jobz` is `'O'` and m is greater than or equal to n) or a matrix of the same type as `A`. The argument `Vt` can be `None` (if `jobz` is `'N'` or `'A'` or `jobz` is `'O'` and m is less than n) or a matrix of the same type as `A`.

On exit, the contents of `A` are destroyed.

4.9 Example: Analytic Centering

The analytic centering problem is defined as

$$\text{minimize} \quad -\sum_{i=1}^{m} \log(b_i - a_i^T x).$$

In the code below we solve the problem using Newton's method. At each iteration the Newton direction is computed by solving a positive definite set of linear equations

$$A^T \mathbf{diag}\,(b - Ax)^{-2} A v = -\mathbf{diag}\,(b - Ax)^{-1}\mathbf{1}$$

(where A has rows a_i^T), and a suitable step size is determined by a backtracking line search.
We use the level-3 BLAS function `syrk()` to form the Hessian matrix and the LAPACK function `posv()` to solving the Newton system. The code can be further optimized by replacing the matrix-vector products with the level-2 BLAS function `gemv()`.

```
from cvxopt.base import matrix, log, mul, div
from cvxopt import blas, lapack, random
from math import sqrt

def acent(A,b):
```

```
"""
Returns the analytic center of A*x <= b.
We assume that b > 0 and the feasible set is bounded.
"""

MAXITERS = 100
ALPHA = 0.01
BETA = 0.5
TOL = 1e-8

m, n = A.size
x = matrix(0.0, (n,1))
H = matrix(0.0, (n,n))

for iter in xrange(MAXITERS):

    # Gradient is g = A^T * (1./(b-A*x)).
    d = (b-A*x)**-1
    g = A.T * d

    # Hessian is H = A^T * diag(d)^2 * A.
    Asc = mul( d[:,n*[0]], A )
    blas.syrk(Asc, H, trans='T')

    # Newton step is v = -H^-1 * g.
    v = -g
    lapack.posv(H, v)

    # Terminate if Newton decrement is less than TOL.
    lam = blas.dot(g, v)
    if sqrt(-lam) < TOL: return x

    # Backtracking line search.
    y = mul(A*v, d)
    step = 1.0
    while 1-step*max(y) < 0: step *= BETA
    while True:
        if -sum(log(1-step*y)) < ALPHA*step*lam: break
        step *= BETA
    x += step*v
```

Chapter 5

Discrete Transforms (`cvxopt.fftw`)

The `cvxopt.fftw` module is an interface to the FFTW library and contains routines for discrete Fourier, cosine, and sine transforms. This module is optional, and only installed when the FFTW library is made available during the CVXOPT installation.
See also: FFTW3 code, documentation, copyright and license.[1]

5.1 Discrete Fourier Transform

dft(X)

Replaces the columns of a dense complex matrix with their discrete Fourier transforms: if X has *n* rows,

$$X[k,:] := \sum_{j=0}^{n-1} e^{-2\pi jk\sqrt{-1}/n} X[j,:], \qquad k = 0, \ldots, n-1.$$

idft(X)

Replaces the columns of a dense complex matrix with their inverse discrete Fourier transforms: if X has *n* rows,

$$X[k,:] := \frac{1}{n} \sum_{j=0}^{n-1} e^{2\pi jk\sqrt{-1}/n} X[j,:], \qquad k = 0, \ldots, n-1.$$

The separable discrete two dimensional Fourier transform first computes the corresponding one dimensional tranform along the columns of the matrix, followed by the one dimensional transform along the rows of the matrix.
dft2(X)

Replaces a dense complex matrix with the two dimensional discrete Fourier transform.

idft2(X)

Replaces a dense complex matrix with the inverse two dimensional discrete Fourier transform.

5.2 Discrete Cosine Transform

dct(X[, type=2])

[1] `http://www.fftw.org`

Replaces the columns of a dense real matrix with their discrete cosine transforms. The second argument, an integer between 1 and 4, denotes the type of transform (DCT-I, DCT-II, DCT-III, DCT-IV). The DCT-I transform requires that the row dimension of `X` is at least 2. These transforms are defined as follows (for a matrix with *n* rows).

$$\text{DCT-I:} \quad X[k,:] := X[0,:] + (-1)^k X[n-1,:] + 2\sum_{j=1}^{n-2} X[j,:]\cos(\pi jk/(n-1)), \qquad k = 0,\ldots,n-1.$$

$$\text{DCT-II:} \quad X[k,:] := 2\sum_{j=0}^{n-1} X[j,:]\cos(\pi(j+1/2)k/n), \qquad k = 0,\ldots,n-1.$$

$$\text{DCT-III:} \quad X[k,:] := X[0,:] + 2\sum_{j=1}^{n-1} X[j,:]\cos(\pi j(k+1/2)/n), \qquad k = 0,\ldots,n-1.$$

$$\text{DCT-IV:} \quad X[k,:] := 2\sum_{j=0}^{n-1} X[j,:]\cos(\pi(j+1/2)(k+1/2)/n), \qquad k = 0,\ldots,n-1.$$

idct(`X[, type=2]`)

Replaces the columns of a dense real matrix with the inverses of the discrete cosine transforms defined above.

The separable discrete two dimensional cosine transform first computes the corresponding one dimensional tranform along the columns of the matrix, followed by the one dimensional transform along the rows of the matrix.
dct2(`X[, type=2]`)

Replaces a dense real matrix with the two dimensional discrete cosine transform.

idct2(`X[, type=2]`)

Replaces a dense real matrix with the inverse two dimensional discrete cosine transform.

5.3 Discrete Sine Transform

dst(`X[, type=1]`)

Replaces the columns of a dense real matrix with their discrete sine transforms. The second argument, an integer between 1 and 4, denotes the type of transform (DST-I, DST-II, DST-III, DST-IV). These transforms are defined as follows (for a matrix with *n* rows).

$$\text{DST-I:} \quad X[k,:] := 2\sum_{j=0}^{n-1} X[j,:]\sin(\pi(j+1)(k+1)/(n+1)), \qquad k = 0,\ldots,n-1.$$

$$\text{DST-II:} \quad X[k,:] := 2\sum_{j=0}^{n-1} X[j,:]\sin(\pi(j+1/2)(k+1)/n), \qquad k = 0,\ldots,n-1.$$

$$\text{DST-III:} \quad X[k,:] := (-1)^k X[n-1,:] + 2\sum_{j=0}^{n-2} X[j,:]\sin(\pi(j+1)(k+1/2)/n), \qquad k = 0,\ldots,n-1.$$

$$\text{DST-IV:} \quad X[k,:] := 2\sum_{j=0}^{n-1} X[j,:]\sin(\pi(j+1/2)(k+1/2)/n), \qquad k = 0,\ldots,n-1.$$

idst(`X[, type=1]`)

Replaces the columns of a dense real matrix with the inverses of the discrete sine transforms defined above.

The separable discrete two dimensional sine transform first computes the corresponding one dimensional tranform along the columns of the matrix, followed by the one dimensional transform along the rows of the matrix.

dst2(`X[, type=1]`)

Replaces a dense real matrix with the two dimensional discrete sine transform.

idst2(`X[, type=1]`)

Replaces a dense real matrix with the inverse two dimensional discrete sine transform.

Chapter 6

Sparse Matrices (`cvxopt.base`)

In this chapter we discuss the `spmatrix` object defined in `cvxopt.base`.

6.1 Creating Sparse Matrices

A general `spmatrix` object can be thought of as a *triplet description* of a sparse matrix, i.e., a list of entries of the matrix, with for each entry the value, row index, and column index. Entries that are not included in the list are assumed to be zero. For example, the sparse matrix

$$A = \begin{bmatrix} 0 & 2 & 0 & 0 & 3 \\ 2 & 0 & 0 & 0 & 0 \\ -1 & -2 & 0 & 4 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{6.1}$$

has the triplet description

$$(2,1,0), \quad (-1,2,0), \quad (2,0,1), \quad (-2,2,1), \quad (1,3,2), \quad (4,2,3), \quad (3,0,4).$$

The list may include entries with a zero value, so triplet descriptions are not necessarily unique. The list

$$(2,1,0), \quad (-1,2,0), \quad (0,3,0), \quad (2,0,1), \quad (-2,2,1), \quad (1,3,2), \quad (4,2,3), \quad (3,0,4)$$

is another triplet description of the same matrix.
An `spmatrix` object corresponds to a particular triplet description of a sparse matrix. We will refer to the entries in the triplet description as the *nonzero entries* of the object, even though they may have a numerical value zero.
Two functions are provided to create sparse matrices. The first, `spmatrix()`, constructs a sparse matrix from a triplet description.

spmatrix(`x, I, J[, size[, tc]]`)

> `I` and `J` are sequences of integers (lists, tuples, `array` arrays, xrange objects, ...) or integer matrices (`matrix` objects with typecode `'i'`), containing the row and column indices of the nonzero entries. The lengths of `I` and `J` must be equal. If they are matrices, they are treated as lists of indices stored in column-major order, i.e., as lists `list(I)`, respectively, `list(J)`.
>
> `size` is a tuple of nonnegative integers with the row and column dimensions of the matrix. The `size` argument is only needed when creating a matrix with a zero last row or last column. If `size` is not specified, it is determined from `I` and `J`: the default value for `size[0]` is max(`I`)+1 if `I` is nonempty and zero otherwise. The default value for `size[1]` is max(`J`)+1 if `J` is nonempty and zero otherwise.
>
> `tc` is the typecode, `'d'` or `'z'`, for double and complex matrices, respectively. Integer sparse matrices are not implemented.

`x` can be a number, a sequence of numbers, or a dense matrix. This argument specifies the numerical values of the nonzero entries.

- If `x` is a number (Python `integer`, `float` or `complex`), a matrix is created with the sparsity pattern defined by `I` and `J`, and nonzero entries initialized to the value of `x`. The default value of `tc` is `'d'` if `x` is `integer` or `float`, and `'z'` if `x` is `complex`.

  The following code creates a 4 by 4 sparse identity matrix.

  ```
  >>> from cvxopt.base import spmatrix
  >>> A = spmatrix(1.0, range(4), range(4))
  >>> print A
  [ 1.00e+00     0         0         0    ]
  [    0      1.00e+00     0         0    ]
  [    0         0      1.00e+00     0    ]
  [    0         0         0      1.00e+00]
  ```

- If `x` is a sequence of numbers, a sparse matrix is created with the entries of `x` copied to the entries indexed by `I` and `J`. The list `x` must have the same length as `I` and `J`. The default value of `tc` is determined from the elements of `x`: `'d'` if `x` contains integers and floating-point numbers or if `x` is an empty list, and `'z'` if `x` contains at least one complex number.

  As an example, the matrix (6.1) can be created as follows.

  ```
  >>> A = spmatrix([2,-1,2,-2,1,4,3], [1,2,0,2,3,2,0], [0,0,1,1,2,3,4])
  >>> print A
  [    0      2.00e+00     0         0      3.00e+00]
  [ 2.00e+00     0         0         0         0    ]
  [-1.00e+00 -2.00e+00     0      4.00e+00     0    ]
  [    0         0      1.00e+00     0         0    ]
  ```

- If `x` is a dense matrix, a sparse matrix is created with all the entries of `x` copied, in column-major order, to the entries indexed by `I` and `J`. The matrix `x` must have the same length as `I` and `J`. The default value of `tc` is `'d'` if `x` is an `'i'` or `'d'` matrix, and `'z'` otherwise.

If `I` and `J` contain repeated entries, the corresponding values of the coefficients are added.

The function `sparse()` constructs a sparse matrix from a block-matrix description.
sparse(`x[, tc]`)

`tc` is the typecode, `'d'` or `'z'`, for double and complex matrices, respectively.

`x` can be a `matrix`, `spmatrix`, or a list of lists of matrices (`matrix` or `spmatrix` objects) and numbers (Python `integer`, `float` or `complex`).

- If `x` is a `matrix` or `spmatrix` object, then a sparse matrix of the same size and the same numerical value is created. Numerical zeros in `x` are treated as structural zeros and removed from the triplet description of the new sparse matrix.
- If `x` is a list of lists of matrices (`matrix` or `spmatrix`) and numbers (Python `integer`, `float` or `complex`) then each element of `x` is interpreted as a (block-)column matrix stored in colum-major order, and a block-matrix is constructed by juxtaposing the `len(x)` block-columns (as in `matrix()`, see section 2.1). Numerical zeros are removed from the triplet description of the new matrix.

  The following example shows how to construct a sparse block-matrix.

  ```
  >>> from cvxopt.base import matrix, spmatrix, sparse
  >>> A = matrix([[1, 2, 0], [2, 1, 2], [0, 2, 1]])
  >>> B = spmatrix([], [], [], (3,3))
  >>> C = spmatrix([3, 4, 5], [0, 1, 2], [0, 1, 2])
  ```

```
>>> D = sparse([[A, B], [B, C]])
>>> print D
[ 1.00e+00  2.00e+00     0         0         0         0    ]
[ 2.00e+00  1.00e+00  2.00e+00     0         0         0    ]
[    0      2.00e+00  1.00e+00     0         0         0    ]
[    0         0         0      3.00e+00     0         0    ]
[    0         0         0         0      4.00e+00     0    ]
[    0         0         0         0         0      5.00e+00]
```

A matrix with a single block-column can be represented by a single list.

```
>>> D = sparse([A, C])
>>> print D
[ 1.00e+00  2.00e+00     0    ]
[ 2.00e+00  1.00e+00  2.00e+00]
[    0      2.00e+00  1.00e+00]
[ 3.00e+00     0         0    ]
[    0      4.00e+00     0    ]
[    0         0      5.00e+00]
```

The function `spdiag()` constructs a block-diagonal sparse matrix from a list of matrices.

spdiag(`x`)

`x` is a matrix with a single row or column, or a list of square dense or sparse matrices or scalars. If `x` is matrix, a sparse diagonal matrix is returned with the entries of `x` on its diagonal. If `x` is list, a sparse block-diagonal matrix is returned with the element in the list as its diagonal blocks.

The following example shows how to construct a sparse block-diagonal matrix.

```
>>> from cvxopt.base import matrix, spmatrix, spdiag
>>> A = 3.0
>>> B = matrix([[1,-2],[-2,1]])
>>> C = spmatrix([1,1,1,1,1],[0,1,2,0,0,],[0,0,0,1,2])
>>> D = spdiag([A, B, C])
>>> print D
[ 3.00e+00     0         0         0         0         0    ]
[    0      1.00e+00 -2.00e+00     0         0         0    ]
[    0     -2.00e+00  1.00e+00     0         0         0    ]
[    0         0         0      1.00e+00  1.00e+00  1.00e+00]
[    0         0         0      1.00e+00     0         0    ]
[    0         0         0      1.00e+00     0         0    ]
```

6.2 Attributes and Methods

The following attributes and methods are defined for `spmatrix` objects.

V

A single-column dense matrix containing the numerical values of the nonzero entries in column-major order. Making an assignment to the attribute is an efficient way of changing the values of the sparse matrix, without changing the sparsity pattern.

When the attribute `V` is read, a *copy* of `V` is returned, as a new dense matrix. (This implies, for example, that an indexed assignment "`A.V[I] = B`" does not work, or at least cannot be used to modify `A`. Instead the attribute `V` will be read and returned as a new matrix; then the elements of this new matrix are modified.)

I

A single-column integer matrix with the row indices of the entries in `V`. A read-only attribute.

J

A single-column integer matrix with the column indices of the entries in `V`. A read-only attribute.

size

A tuple with the dimensions of the matrix. The size of the matrix can be changed by altering this attribute, as long as the number of elements in the matrix remains unchanged.

CCS

A triplet (colptr, rowind, values) with the compressed-column-storage representation of the matrix. A read-only attribute. This attribute can be used to export sparse matrices to other packages such as MOSEK.

trans()

Returns the transpose of a sparse matrix as a new sparse matrix. One can also use `A.T` instead of `A.trans()`.

ctrans()

Returns the complex conjugate transpose of a sparse matrix as a new sparse matrix. One can also use `A.H` instead of `A.ctrans()`.

In the following example we take the elementwise square root of the matrix

$$A = \begin{bmatrix} 0 & 2 & 0 & 0 & 3 \\ 2 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 4 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{6.2}$$

```
>>> from cvxopt.base import sqrt
>>> A = spmatrix([2,1,2,2,1,3,4], [1,2,0,2,3,0,2], [0,0,1,1,2,3,3])
>>> B = spmatrix(sqrt(A.V), A.I, A.J)
>>> print B
[    0      1.41e+00     0      1.73e+00]
[ 1.41e+00     0          0         0    ]
[ 1.00e+00  1.41e+00      0      2.00e+00]
[    0          0      1.00e+00     0    ]
```

The next example below illustrates assignments to `V`.

```
>>> from cvxopt.base import spmatrix, matrix
>>> A = spmatrix(range(5), [0,1,1,2,2], [0,0,1,1,2])
>>> print A
[ 0.00e+00     0          0    ]
[ 1.00e+00  2.00e+00      0    ]
[    0      3.00e+00  4.00e+00]
>>> B = spmatrix(A.V, A.J, A.I, (4,4))  # transpose and add a zero row and column
>>> print B
[ 0.00e+00  1.00e+00      0          0    ]
[    0      2.00e+00  3.00e+00      0    ]
[    0          0      4.00e+00      0    ]
[    0          0          0         0    ]
>>> print matrix(B)
[ 0.00e+00  1.00e+00  0.00e+00  0.00e+00]
```

```
[ 0.00e+00  2.00e+00  3.00e+00  0.00e+00]
[ 0.00e+00  0.00e+00  4.00e+00  0.00e+00]
[ 0.00e+00  0.00e+00  0.00e+00  0.00e+00]
>>> B.V = matrix([1., 7., 8., 6., 4.])   # assign new values to nonzero entries
>>> print B
[ 1.00e+00  7.00e+00     0         0    ]
[    0      8.00e+00  6.00e+00     0    ]
[    0         0      4.00e+00     0    ]
[    0         0         0         0    ]
>>> B.V += 1.0   # add 1 to the nonzero entries
>>> print B
[ 2.00e+00  8.00e+00     0         0    ]
[    0      9.00e+00  7.00e+00     0    ]
[    0         0      5.00e+00     0    ]
[    0         0         0         0    ]
```

The `V`, `I` and `J` attributes can be used for reading sparse matrices from or writing them to binary files. Suppose we want to write the matrix `A` defined above to a binary file.

```
>>> f = open('test.bin','w')
>>> A.V.tofile(f)
>>> A.I.tofile(f)
>>> A.J.tofile(f)
>>> f.close()
```

A sparse matrix can be created from this file as follows.

```
>>> f = open('test.bin','r')
>>> V = matrix(0.0, (5,1));  V.fromfile(f)
>>> I = matrix(0, (5,1));  I.fromfile(f)
>>> J = matrix(0, (5,1));  J.fromfile(f)
>>> B = spmatrix(V, I, J)
>>> print B
[ 0.00e+00     0         0    ]
[ 1.00e+00  2.00e+00     0    ]
[    0      3.00e+00  4.00e+00]
```

Note that the `pickle` module provides a convenient alternative to this method.

6.3 Arithmetic Operations

Most of the operations defined for dense `'d'` and `'z'` matrices (section 2.3) are also defined for sparse matrices. In the following table, `A` is a sparse matrix, `B` is sparse or dense, and `c` is a scalar, defined as a Python number or a 1 by 1 dense matrix.

Unary plus/minus	`+A`, `-A`
Addition	`A+B`, `B+A`, `A+c`, `c+A`
Subtraction	`A-B`, `B-A`, `A-c`, `c-A`
Matrix multiplication	`A*B`, `B*A`
Scalar multiplication and division	`c*A`, `A*c`, `A/c`

If `B` is a dense matrix, then the result of `A+B`, `B+A`, `A-B`, `B-A` is a dense matrix. The typecode of the result is `'d'` if `A` has typecode `'d'` and `B` has typecode `'i'` or `'d'`, and it is `'z'` if `A` and/or `B` have typecode `'z'`.

If B is a sparse matrix, then the result of A+B, B+A, A-B, B-A is a sparse matrix. The typecode of the result is 'd' if A and B have typecode 'd', and 'z' otherwise.
If c in A+c, A-c, c+A, c-A is a number, then it is interpreted as a dense matrix with the same size as A, typecode given by the type of c, and all entries equal to c. If c is a 1 by 1 dense matrix and the size of A is not 1 by 1, then c is interpreted as a dense matrix of the same size as A, typecode given by the typecode of c, and all entries equal to c[0].
The result of a matrix-matrix product A*B or B*A is a dense matrix if B is dense, and sparse if B is sparse. The matrix-matrix product is not allowed if B is a dense 'i' matrix.
If c is a number (Python integer float or complex), then the operations c*A and A*c define scalar multiplication and return a sparse matrix.
If c is a 1 by 1 dense matrix, then, if possible, the products c*A and A*c are interpreted as matrix-matrix products and a dense matrix is returned. If the product cannot be interpreted as a matrix-matrix product (either because the dimensions of A are incompatible or because c has typecode 'i'), then the product is interpreted as the scalar multiplication with c[0] and a sparse matrix is returned.
The division A/c is interpreted as scalar multiplication with 1.0/c if c is a number, or with 1.0/c[0] if c is a 1 by 1 dense matrix.
The following in-place operations are defined for a sparse matrix A if they do not change the dimensions or type of A.

In-place addition	A+=B, A+=c
In-place subtraction	A-=B, A-=c
In-place scalar multiplication and division	A*=c, A/=c

For example, "A += 1.0" is not allowed because the operation "A = A + 1.0" results in a dense matrix, so it cannot be assigned to A without changing its type.
In-place matrix-matrix products are not allowed. (Except when c is a 1 by 1 dense matrix, in which case A*=c is interpreted as a scalar product A*=c[0].)
As for dense operations, the in-place sparse operations do not return a new matrix but modify the existing object A.

6.4 Indexing and Slicing

Sparse matrices can be indexed the same way as dense matrices (see section 2.4).

```
>>> from cvxopt.base import spmatrix
>>> A = spmatrix([0,2,-1,2,-2,1], [0,1,2,0,2,1], [0,0,0,1,1,2])
>>> print A[:,[0,1]]
[ 0.00e+00  2.00e+00]
[ 2.00e+00     0    ]
[-1.00e+00 -2.00e+00]
>>> B = spmatrix([0,2*1j,0,-2], [1,2,1,2], [0,0,1,1,])
>>> print B[-2:,-2:]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00]
[ 0.00e+00+j2.00e+00 -2.00e+00-j0.00e+00]
```

An indexed sparse matrix A[I] or A[I,J] can also be the target of an assignment. The righthand side of the assignment can be a scalar (a Python integer, float, or complex, or a 1 by 1 dense matrix), a sequence of numbers, or a sparse or dense matrix of compatible dimensions. If the righthand side is a scalar, it is treated as a dense matrix of the same size as the lefthand side and with all its entries equal to the scalar. If the righthand side is a sequence of numbers, they are treated as the elements of a dense matrix in column-major order.
We continue the example above.

```
>>> C = spmatrix([10,-20,30], [0,2,1], [0,0,1])
>>> A[:,0] = C[:,0]
>>> print A
```

```
[ 1.00e+01  2.00e+00     0    ]
[    0         0      1.00e+00]
[-2.00e+01 -2.00e+00     0    ]
>>> D = matrix(range(6), (3,2))
>>> A[:,0] = D[:,0]
>>> print A
[ 0.00e+00  2.00e+00     0    ]
[ 1.00e+00     0      1.00e+00]
[ 2.00e+00 -2.00e+00     0    ]
>>> A[:,0] = 1
>>> print A
[ 1.00e+00  2.00e+00     0    ]
[ 1.00e+00     0      1.00e+00]
[ 1.00e+00 -2.00e+00     0    ]
>>> A[:,0] = 0
>>> print A
[ 0.00e+00  2.00e+00     0    ]
[ 0.00e+00     0      1.00e+00]
[ 0.00e+00 -2.00e+00     0    ]
```

6.5 Built-In Functions

The functions described in the table of section 2.5 also work with sparse matrix arguments. The difference is that for a sparse matrix only the nonzero entries are considered.

len(x)

If x is a spmatrix, returns the number of nonzero entries in x.

bool([x])

If x is a spmatrix, returns False if x has at least one nonzero entry; False otherwise.

max(x)

If x is a spmatrix, returns the maximum nonzero entry of x.

min(x)

If x is a spmatrix, returns the minimum nonzero entry of x.

abs(x)

If x is a spmatrix, returns a sparse matrix with the absolute value of the elements of x and the same sparsity pattern.

sum(x[, start=0.0])

If x is a spmatrix, returns the sum of start and the elements of x.

The functions list(), tuple(), zip(), map(), filter() also take sparse matrix arguments. They work as for dense matrices, again with the difference that only the nonzero entries are considered.
In the following example we square the entries of the matrix (6.2).

```
>>> A = spmatrix([2,1,2,2,1,3,4], [1,2,0,2,3,0,2], [0,0,1,1,2,3,3])
>>> B = spmatrix(map(lambda x: x**2, A), A.I, A.J)
>>> print B
```

```
[    0       4.00e+00      0       9.00e+00]
[ 4.00e+00      0          0          0    ]
[ 1.00e+00   4.00e+00      0       1.60e+01]
[    0          0       1.00e+00      0    ]
```

The expression "`x in A`" returns `True` if a nonzero entry of `A` is equal to `x` and `False` otherwise.

6.6 Sparse BLAS Functions

The `cvxopt.base` module includes a few arithmetic functions that extend functions from `cvxopt.blas` to sparse matrices. These functions are faster than the corresponding operations implemented using the overloaded arithmetic described in section 6.3. They also work in-place, i.e., they modify their arguments without creating new objects.

gemv(`A, x, y[, trans='N'[, alpha=1.0[, beta=0.0]]]`)

Matrix-vector product with a general dense or sparse matrix:

$$y := \alpha Ax+\beta y \quad (\text{trans} = \text{'N'}), \qquad y := \alpha A^T x+\beta y \quad (\text{trans} = \text{'T'}), \qquad y := \alpha A^H x+\beta y \quad (\text{trans} = \text{'C'}).$$

If `A` is a dense matrix, this is identical to `blas.gemv()`. If `A` is sparse, the result is the same as when `blas.gemv()` is called with `matrix(A)` as argument, however, without explicitly converting `A` to dense.

symv(`A, x, y[, uplo='L'[, alpha=1.0[, beta=0.0]]]`)

Matrix-vector product with a dense or sparse real symmetric matrix:

$$y := \alpha Ax + \beta y.$$

If `A` is a dense matrix, this is identical to `blas.symv()`. If `A` is sparse, the result is the same as when `blas.symv()` is called with `matrix(A)` as argument, however, without explicitly converting `A` to dense.

gemm(`A, B, C[, transA='N'[, transB='N'[, alpha=1.0[, beta=0.0[, partial=False]]]]]`)

Matrix-matrix product of two general sparse or dense matrices:

$$C := \alpha\,\text{op}(A)\,\text{op}(B) + \beta C$$

where

$$\text{op}(A) = \begin{cases} A & \text{transA} = \text{'N'} \\ A^T & \text{transA} = \text{'T'} \\ A^H & \text{transA} = \text{'C'} \end{cases} \qquad \text{op}(B) = \begin{cases} B & \text{transB} = \text{'N'} \\ B^T & \text{transB} = \text{'T'} \\ B^H & \text{transB} = \text{'C'}. \end{cases}$$

If `A`, `B` and `C` are dense matrices, this is identical to `blas.gemm()`, described in section 3.4, and the argument `partial` is ignored.

If `A` and/or `B` are sparse and `C` is dense, the result is the same as when `blas.gemm()` is called with `matrix(A)` and `matrix(B)` as arguments, without explicitly converting `A` and `B` to dense. The argument `partial` is ignored.

If `C` is a sparse matrix, the matrix-matrix product in the definition of `blas.gemm()` is computed, but as a sparse matrix. If `partial` is `False`, the result is stored in `C`, and the sparsity pattern of `C` is modified if necessary. If `partial` is `True`, the operation only updates the nonzero elements in `C`, even if the sparsity pattern of `C` differs from that of the matrix product.

syrk(`A, C[, uplo='L'[, trans='N'[, alpha=1.0[, beta=0.0[, partial=False]]]]]`)

Rank-k update of a sparse or dense real or complex symmetric matrix:

$$C := \alpha AA^T + \beta C \quad (\text{trans} = \text{'N'}), \qquad C := \alpha A^T A + \beta C \quad (\text{trans} = \text{'T'}),$$

If `A` and `C` are dense, this is identical to `blas.syrk()`, described in section 3.4, and the argument `partial` is ignored.

If `A` is sparse and `C` is dense, the result is the same as when `blas.syrk()` is called with `matrix(A)` as argument, without explicitly converting `A` to dense. The argument `partial` is ignored.

If `C` is sparse, the product in the definition of `blas.syrk()` is computed, but as a sparse matrix. If `partial` is `False`, the result is stored in `C`, and the sparsity pattern of `C` is modified if necessary. If `partial` is `True`, the operation only updates the nonzero elements in `C`, even if the sparsity pattern of `C` differs from that of the matrix product.

In the following example, we first compute

$$C = A^T B, \qquad A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 0 & -1 & 0 \\ 2 & 0 & 2 \\ 0 & 3 & 0 \\ 2 & 0 & 0 \end{bmatrix}.$$

```
>>> from cvxopt.base import spmatrix, gemm
>>> A = spmatrix(1, [1,3,0,2,1], [0,0,1,1,2])
>>> B = spmatrix([2,2,-1,3,2], [1,3,0,2,1], [0,0,1,1,2])
>>> C = spmatrix([], [], [], size=(3,3))
>>> gemm(A, B, C, transA='T')
>>> print C
[ 4.00e+00     0      2.00e+00]
[    0      2.00e+00     0    ]
[ 2.00e+00     0      2.00e+00]
```

Now suppose we want to replace `C` with

$$C = A^T D, \qquad D = \begin{bmatrix} 0 & 1 & 0 \\ 3 & 0 & -2 \\ 0 & 1 & 0 \\ 4 & 0 & 0 \end{bmatrix}.$$

The new matrix has the same sparsity pattern as `C`, so we can use `gemm()` with the `partial=True` option. This saves time in large sparse matrix multiplications when the sparsity pattern of the result is known beforehand.

```
>>> D = spmatrix([3,4,1,1,-2], [1,3,0,2,1], [0,0,1,1,2])
>>> gemm(A, D, C, transA='T', partial=True)
>>> print C
[ 7.00e+00     0     -2.00e+00]
[    0      2.00e+00     0    ]
[ 3.00e+00     0     -2.00e+00]
```

Chapter 7

Sparse Linear Equations

In this section we describe routines for solving sparse sets of linear equations.
A real symmetric or complex Hermitian sparse matrix is stored as an `spmatrix` object `X` of size (n, n) and an additional character argument `uplo` with possible values `'L'` and `'U'`. If `uplo` is `'L'`, the lower triangular part of `X` contains the lower triangular part of the symmetric or Hermitian matrix, and the upper triangular matrix of `X` is ignored. If `uplo` is `'U'`, the upper triangular part of `X` contains the upper triangular part of the matrix, and the lower triangular matrix of `X` is ignored.
A general sparse square matrix of order n is represented by an `spmatrix` object of size (n, n).
Dense matrices, which appear as righthand sides of equations, are stored using the same conventions as in the BLAS and LAPACK modules.

7.1 Matrix Orderings (`cvxopt.amd`)

CVXOPT includes an interface to the AMD library for computing approximate minimum degree orderings of sparse matrices.
See also:

- AMD code, documentation, copyright and license.[1]
- P. R. Amestoy, T. A. Davis, I. S. Duff, Algorithm 837: AMD, An Approximate Minimum Degree Ordering Algorithm, ACM Transactions on Mathematical Software, 30(3), 381-388, 2004.

order(`A[, uplo='L']`)

Computes the approximate mimimum degree ordering of a symmetric sparse matrix A. The ordering is returned as an integer dense matrix with length equal to the order of `A`. Its entries specify a permutation that reduces fill-in during the Cholesky factorization. More precisely, if `p = order(A)`, then `A[p,p]` has sparser Cholesky factors than `A`.

As an example we consider the matrix

$$\begin{bmatrix} 10 & 0 & 3 & 0 \\ 0 & 5 & 0 & -2 \\ 3 & 0 & 5 & 0 \\ 0 & -2 & 0 & 2 \end{bmatrix}.$$

```
>>> from cvxopt.base import spmatrix
>>> from cvxopt import amd
>>> A = spmatrix([10,3,5,-2,5,2], [0,2,1,2,2,3], [0,0,1,1,2,3])
```

[1] `http://www.cise.ufl.edu/research/sparse/amd`

```
>>> P = amd.order(A)
>>> print P
[ 1]
[ 0]
[ 2]
[ 3]
```

7.2 General Linear Equations (`cvxopt.umfpack`)

The module `cvxopt.umfpack` includes four functions for solving sparse non-symmetric sets of linear equations. They call routines from the UMFPACK library, with all control options set to the default values described in the UMFPACK user guide.
See also:

- UMFPACK code, documentation, copyright and license.[2]
- T. A. Davis, Algorithm 832: UMFPACK – an unsymmetric-pattern multifrontal method with a column pre-ordering strategy, ACM Transactions on Mathematical Software, 30(2), 196-199, 2004.

linsolve(`A, B[, trans='N']`)

Solves a sparse set of linear equations

$$AX = B \quad (\text{trans} = \text{'N'}), \qquad A^T X = B \quad (\text{trans} = \text{'T'}), \qquad A^H X = B \quad (\text{trans} = \text{'C'}),$$

where `A` is a sparse matrix and `B` is a dense matrix of the same type (`'d'` or `'z'`) as `A`. On exit `B` contains the solution. Raises an `ArithmeticError` exception if the coefficient matrix is singular.

In the following example we solve an equation with coefficient matrix

$$A = \begin{bmatrix} 2 & 3 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 6 \\ 0 & -1 & -3 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 4 & 2 & 0 & 1 \end{bmatrix}. \tag{7.1}$$

```
>>> from cvxopt.base import spmatrix, matrix
>>> from cvxopt import umfpack
>>> V = [2,3, 3,-1,4, 4,-3,1,2, 2, 6,1]
>>> I = [0,1, 0, 2,4, 1, 2,3,4, 2, 1,4]
>>> J = [0,0, 1, 1,1, 2, 2,2,2, 3, 4,4]
>>> A = spmatrix(V,I,J)
>>> B = matrix(1.0, (5,1))
>>> umfpack.linsolve(A,B)
>>> print B
[ 5.79e-01]
[-5.26e-02]
[ 1.00e+00]
[ 1.97e+00]
[-7.89e-01]
```

The function `umfpack.linsolve()` is equivalent to the following three functions called in sequence.
symbolic(`A`)

[2] `http://www.cise.ufl.edu/research/sparse/umfpack`

Reorders the columns of `A` to reduce fill-in and performs a symbolic LU factorization. `A` is a sparse, possibly rectangular, matrix. Returns the symbolic factorization as an opaque C object that can be passed on to `umfpack.numeric()`.

numeric(`A, F`)

Performs a numeric LU factorization of a sparse, possibly rectangular, matrix `A`. The argument `F` is the symbolic factorization computed by `umfpack.symbolic()` applied to the matrix `A`, or another sparse matrix with the same sparsity pattern, dimensions, and type. The numeric factorization is returned as an opaque C object that that can be passed on to `umfpack.solve()`. Raises an `ArithmeticError` if the matrix is singular.

solve(`A, F, B[, trans='N']`)

Solves a set of linear equations

$$AX = B \quad (\text{trans} = \text{'N'}), \qquad A^T X = B \quad (\text{trans} = \text{'T'}), \qquad A^H X = B \quad (\text{trans} = \text{'C'}),$$

where `A` is a sparse matrix and `B` is a dense matrix of the same type as `A`. The argument `F` is a numeric factorization computed by `umfpack.numeric()`. On exit `B` is overwritten by the solution.

These separate functions are useful for solving several sets of linear equations with the same coefficient matrix and different righthand sides, or with coefficient matrices that share the same sparsity pattern. The symbolic factorization depends only on the sparsity pattern of the matrix, and not on the numerical values of the nonzero coefficients. The numerical factorization on the other hand depends on the sparsity pattern of the matrix and on its the numerical values. As an example, suppose A is the matrix (7.1) and

$$B = \begin{bmatrix} 4 & 3 & 0 & 0 & 0 \\ 3 & 0 & 4 & 0 & 6 \\ 0 & -1 & -3 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 4 & 2 & 0 & 2 \end{bmatrix},$$

which differs from A in its first and last entries. The following code computes

$$x = A^{-T} B^{-1} A^{-1} \mathbf{1}.$$

```
>>> from cvxopt.base import spmatrix, matrix
>>> from cvxopt import umfpack
>>> VA = [2,3, 3,-1,4, 4,-3,1,2, 2, 6,1]
>>> VB = [4,3, 3,-1,4, 4,-3,1,2, 2, 6,2]
>>> I =  [0,1, 0, 2,4, 1, 2,3,4, 2, 1,4]
>>> J =  [0,0, 1, 1,1, 2, 2,2,2, 3, 4,4]
>>> A = spmatrix(VA, I, J)
>>> B = spmatrix(VB, I, J)
>>> x = matrix(1.0, (5,1))
>>> Fs = umfpack.symbolic(A)
>>> FA = umfpack.numeric(A, Fs)
>>> FB = umfpack.numeric(B, Fs)
>>> umfpack.solve(A, FA, x)
>>> umfpack.solve(B, FB, x)
>>> umfpack.solve(A, FA, x, trans='T')
>>> print x
[ 5.81e-01]
[-2.37e-01]
[ 1.63e+00]
[ 8.07e+00]
[-1.31e-01]
```

7.3 Positive Definite Linear Equations (cvxopt.cholmod)

cvxopt.cholmod is an interface to the Cholesky factorization routines of the CHOLMOD package. It includes functions for Cholesky factorization of sparse positive definite matrices, and for solving sparse sets of linear equations with positive definite matrices. The routines can also be used for computing $\mathrm{LDL}^{\mathrm{T}}$ (or $\mathrm{LDL}^{\mathrm{H}}$) factorizations of symmetric indefinite matrices (with L unit lower-triangular and D diagonal and nonsingular) if such a factorization exists.

See also: CHOLMOD code, documentation, copyright and license.[3]

linsolve(A, B[, p=None[, uplo='L']])

Solves

$$AX = B$$

with A sparse and real symmetric or complex Hermitian. B is a dense matrix of the same type as A. On exit it is overwritten with the solution. The argument p is an integer matrix with length equal to the order of A, and specifies an optional reordering of A. If p is not specified, CHOLMOD used a reordering from the AMD library. Raises an ArithmeticError if the factorization does not exist.

As an example, we solve

$$\begin{bmatrix} 10 & 0 & 3 & 0 \\ 0 & 5 & 0 & -2 \\ 3 & 0 & 5 & 0 \\ 0 & -2 & 0 & 2 \end{bmatrix} X = \begin{bmatrix} 0 & 4 \\ 1 & 5 \\ 2 & 6 \\ 3 & 7 \end{bmatrix}. \tag{7.2}$$

```
>>> from cvxopt.base import matrix, spmatrix
>>> from cvxopt import cholmod
>>> A = spmatrix([10,3, 5,-2, 5, 2], [0,2, 1,3, 2, 3], [0,0, 1,1, 2, 3])
>>> X = matrix(range(8), (4,2), 'd')
>>> cholmod.linsolve(A,X)
>>> print X
[-1.46e-01  4.88e-02]
[ 1.33e+00  4.00e+00]
[ 4.88e-01  1.17e+00]
[ 2.83e+00  7.50e+00]
```

splinsolve(A, B[, p=None[, uplo='L']])

Similar to linsolve() except that B is a sparse matrix and that the solution is returned as an output argument (as a new sparse matrix). B is not modified.

The following code computes the inverse of the coefficient matrix in (7.2) as a sparse matrix.

```
>>> X = cholmod.splinsolve(A, spmatrix(1.0,range(4),range(4)))
>>> print X
[ 1.22e-01     0     -7.32e-02     0    ]
[    0      3.33e-01     0      3.33e-01]
[-7.32e-02     0      2.44e-01     0    ]
[    0      3.33e-01     0      8.33e-01]
```

The functions linsolve() and splinsolve() are equivalent to symbolic() and numeric() called in sequence, followed by solve(), respectively, spsolve().

symbolic(A[, p=None[, uplo='L']])

[3] http://www.cise.ufl.edu/research/sparse/cholmod

Performs a symbolic analysis of a sparse real symmetric or complex Hermitian matrix `A` for one of the two factorizations:

$$PAP^T = LL^T, \qquad PAP^T = LL^H, \tag{7.3}$$

and

$$PAP^T = LDL^T, \qquad PAP^T = LDL^H, \tag{7.4}$$

where P is a permutation matrix, L is lower triangular (unit lower triangular in the second factorization), and D is nonsingular diagonal. The type of factorization depends on the value of `options['supernodal']` (see below).

If `uplo` is `'L'`, only the lower triangular part of `A` is accessed and the upper triangular part is ignored. If `uplo` is `'U'`, only the upper triangular part of `A` is accessed and the lower triangular part is ignored.

If `p` is not provided, a reordering from the AMD library is used.

The symbolic factorization is returned as an opaque C object that can be passed to `cholmod.numeric()`.

numeric(`A, F`)

Performs a numeric factorization of a sparse symmetric matrix as (7.3) or (7.4). The argument `F` is the symbolic factorization computed by `cholmod.symbolic()` applied to the matrix `A`, or to another sparse matrix with the same sparsity pattern and typecode, or by `cholmod.numeric()` applied to a matrix with the same sparsity pattern and typecode as `A`.

If `F` was created by a `cholmod.symbolic` with `uplo` equal to `'L'`, then only the lower triangular part of `A` is accessed and the upper triangular part is ignored. If it was created with `uplo` is `'U'`, then only the upper triangular part of `A` is accessed and the lower triangular part is ignored.

On successful exit, the factorization is stored in `F`. Raises an `ArithmeticError` if the factorization does not exist.

solve(`F, B[, sys=0]`)

Solves one of the following linear equations where `B` is a dense matrix and `F` is the numeric factorization (7.3) or (7.4) computed by `cholmod_numeric()`. `sys` is an integer with values between 0 and 8.

`sys`	equation
0	$AX = B$
1	$LDL^TX = B$
2	$LDLX = B$
3	$DL^TX = B$
4	$LX = B$
5	$L^TX = B$
6	$DX = B$
7	$P^TX = B$
8	$PX = B$

(If `F` is a Cholesky factorization of the form (7.3), D is an identity matrix in this table. If `A` is complex, L^T should be replaced by L^H.)

The matrix `B` is a dense `'d'` or `'z'` matrix, with the same type as `A`. On exit it is overwritten by the solution.

spsolve(`F, B[, sys=0]`)

Similar to `solve()`, except that `B` is a sparse matrix, and the solution is returned as an output argument (as a sparse matrix). `B` must have the same typecode as `A`.

For the same example as above:

```
>>> X = matrix(range(8), (4,2), 'd')
>>> F = cholmod.symbolic(A)
>>> cholmod.numeric(A,F)
>>> cholmod.solve(F,X)
>>> print X
[-1.46e-01  4.88e-02]
[ 1.33e+00  4.00e+00]
[ 4.88e-01  1.17e+00]
[ 2.83e+00  7.50e+00]
```

diag(F)

Returns the diagonal elements of the Cholesky factor L in (7.3), as a dense matrix of the same type as A. Note that this only applies to Cholesky factorizations. The matrix D in an $\mathrm{LDL}^{\mathrm{T}}$ factorization can be retrieved via cholmod.solve() with sys equal to 6.

In the functions listed above, the default values of the control parameters described in the CHOLMOD user guide are used, except for Common->print which is set to 0 instead of 3 and Common->supernodal which is set to 2 instead of 1. These parameters (and a few others) can be modified by making an entry in the dictionary cholmod.options. The meaning of these parameters is as follows.

options['supernodal'] If equal to 0, a factorization (7.4) is computed using a simplicial algorithm. If equal to 2, a factorization (7.3) is computed using a supernodal algorithm. If equal to 1, the most efficient of the two factorizations is selected, based on the sparsity pattern. Default: 2.

options['print'] A nonnegative integer that controls the amount of output printed to the screen. Default: 0 (no output).

As an example that illustrates diag() and the use of cholmod.options, we compute the logarithm of the determinant of the coefficient matrix in (7.2) by two methods.

```
>>> import math
>>> from cvxopt.cholmod import options
>>> from cvxopt.base import log
>>> F = cholmod.symbolic(A)
>>> cholmod.numeric(A,F)
>>> print 2.0 * sum(log(cholmod.diag(F)))
5.50533153593

>>> options['supernodal'] = 0
>>> F = cholmod.symbolic(A)
>>> cholmod.numeric(A,F)
>>> Di = matrix(1.0, (4,1))
>>> cholmod.solve(F,Di,sys=6)
>>> print -sum(log(Di))
5.50533153593
```

7.4 Example: Covariance Selection

This example illustrates the use of the routines for sparse Cholesky factorization. We consider the problem

$$\begin{array}{ll} \text{minimize} & -\log\det K + \mathbf{tr}(KY) \\ \text{subject to} & K_{ij} = 0, \quad (i,j) \notin S. \end{array} \tag{7.5}$$

The optimization variable is a symmetric matrix K of order n and the domain of the problem is the set of positive definite matrices. The matrix Y and the index set S are given. We assume that all the diagonal positions are included in

S. This problem arises in maximum likelihood estimation of the covariance matrix of a zero-mean normal distribution, with constraints that specify that pairs of variables are conditionally independent.
We can express K as

$$K(x) = E_1\mathbf{diag}\,(x)E_2^T + E_2\mathbf{diag}\,(x)E_1^T$$

where x are the nonzero elements in the lower triangular part of K, with the diagonal elements scaled by 1/2, and

$$E_1 = \left[\begin{array}{cccc} e_{i_1} & e_{i_2} & \cdots & e_{i_q} \end{array}\right], \qquad E_2 = \left[\begin{array}{cccc} e_{j_1} & e_{j_2} & \cdots & e_{j_q} \end{array}\right],$$

where $(i_k,\, j_k)$ are the positions of the nonzero entries in the lower-triangular part of K. With this notation, we can solve problem (7.5) by solving the unconstrained problem

$$\text{minimize} \quad f(x) = -\log\det K(x) + \mathbf{tr}(K(x)Y).$$

The code below implements Newton's method with a backtracking line search. The gradient and Hessian of the objective function are given by

$$\begin{array}{rcl} \nabla f(x) & = & 2\mathbf{diag}\,(E_1^T(Y - K(x)^{-1})E_2)) \\ & = & 2\mathbf{diag}\,(Y_{IJ} - \left(K(x)^{-1}\right)_{IJ}) \\ \nabla^2 f(x) & = & 2(E_1^T K(x)^{-1}E_1) \circ (E_2^T K(x)^{-1}E_2) + 2(E_1^T K(x)^{-1}E_2) \circ (E_2^T K(x)^{-1}E_1) \\ & = & 2\left(K(x)^{-1}\right)_{II} \circ \left(K(x)^{-1}\right)_{JJ} + 2\left(K(x)^{-1}\right)_{IJ} \circ \left(K(x)^{-1}\right)_{JI}, \end{array}$$

where $\circ$ denotes Hadamard product.

```
from cvxopt.base import matrix, spmatrix, log, mul
from cvxopt import blas, lapack, amd, cholmod

def covsel(Y):
    """
    Returns the solution of

         minimize    -logdet K + Tr(KY)
         subject to  K_{ij}=0,  (i,j) not in indices listed in I,J.

    Y is a symmetric sparse matrix with nonzero diagonal elements.
    I = Y.I,  J = Y.J.
    """

    I, J = Y.I, Y.J
    n, m = Y.size[0], len(I)
    N = I + J*n         # non-zero positions for one-argument indexing
    D = [k for k in xrange(m) if I[k]==J[k]]  # position of diagonal elements

    # starting point: symmetric identity with nonzero pattern I,J
    K = spmatrix(0, I, J)
    K[::n+1] = 1

    # Kn is used in the line search
    Kn = spmatrix(0, I, J)

    # symbolic factorization of K
    F = cholmod.symbolic(K)
```

```
# Kinv will be the inverse of K
Kinv = matrix(0.0, (n,n))

for iters in xrange(100):

    # numeric factorization of K
    cholmod.numeric(K, F)
    d = cholmod.diag(F)

    # compute Kinv by solving K*X = I
    Kinv[:] = 0
    Kinv[::n+1] = 1
    cholmod.solve(F, Kinv)

    # solve Newton system
    grad = 2*(Y.V - Kinv[N])
    hess = 2*(mul(Kinv[I,J],Kinv[J,I]) + mul(Kinv[I,I],Kinv[J,J]))
    v = -grad
    lapack.posv(hess,v)

    # stopping criterion
    sqntdecr = -blas.dot(grad,v)
    print "Newton decrement squared:%- 7.5e" %sqntdecr
    if (sqntdecr < 1e-12):
        print "number of iterations: ", iters+1
        break

    # line search
    dx = +v
    dx[D] *= 2      # scale the diagonal elems
    f = -2.0 * sum(log(d))    # f = -log det K
    s = 1
    for lsiter in xrange(50):
        Kn.V = K.V + s*dx
        try:
            cholmod.numeric(Kn, F)
        except ArithmeticError:
            s *= 0.5
        else:
            d = cholmod.diag(F)
            fn = -2.0 * sum(log(d)) + 2*s*blas.dot(v,Y.V)
            if (fn < f - 0.01*s*sqntdecr):
                 break
            s *= 0.5

    K.V = Kn.V

return K
```

Chapter 8

Cone Programming (`cvxopt.solvers`)

In this chapter we consider convex optimization problems of the form

$$\begin{array}{ll} \text{minimize} & (1/2)x^T Px + q^T x \\ \text{subject to} & Gx \preceq h \\ & Ax = b. \end{array}$$

The linear inequality is a generalized inequality with respect to a proper convex cone. It may include componentwise vector inequalities, second-order cone inequalities, and linear matrix inequalities. The main solvers are `conelp()` and `coneqp()`, described in sections 8.1 and 8.2. The function `conelp()` is restricted to problems with linear cost functions, but can detect primal and dual infeasibility. The function `coneqp()` solves the general quadratic problem, but requires the problem to be primal and dual feasible. For convenience (and backward compatibility), simpler interfaces to these function are also provided that handle pure linear programs, quadratic programs, second-order cone programs, and semidefinite programs. These are described in sections 8.3–8.6. In section 8.7 we explain how custom solvers can be implemented that exploit structure in cone programs. The last two sections describe optional interfaces to external solvers, and the algorithm parameters that control the cone programming solvers.

8.1 Linear Cone Programs

conelp(`c, G, h[, dims[, A, b[, primalstart[, dualstart[, kktsolver]]]]]`)

Solves a pair of primal and dual cone programs

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Gx + s = h \\ & Ax = b \\ & s \succeq 0 \end{array} \qquad\qquad \begin{array}{ll} \text{maximize} & -h^T z - b^T y \\ \text{subject to} & G^T z + A^T y + c = 0 \\ & z \succeq 0. \end{array}$$

The primal variables are x and the slack variable s. The dual variables are y and z. The inequalities are interpreted as $s \in C$, $z \in C$, where C is a cone defined as a Cartesian product of a nonnegative orthant, a number of second-order cones, and a number of positive semidefinite cones:

$$C = C_0 \times C_1 \times \cdots \times C_M \times C_{M+1} \times \cdots \times C_{M+N}$$

with

$$C_0 = \{u \in \mathbf{R}^l \mid u_k \geq 0,\ k = 1, \ldots, l\}, \qquad C_{k+1} = \{(u_0, u_1) \in \mathbf{R} \times \mathbf{R}^{r_k - 1} \mid u_0 \geq \|u_1\|_2\}, \quad k = 0, \ldots, M-1, \qquad C_{k+M+1}$$

In this definition, $\mathbf{vec}(u)$ denotes a symmetric matrix u stored as a vector in column major order. The structure of C is specified by `dims`. This argument is a dictionary with three fields.

`dims['l']`: l, the dimension of the nonnegative orthant (a nonnegative integer).

`dims['q']`: $[r_0, \ldots, r_{M-1}]$, a list with the dimensions of the second-order cones (positive integers).

`dims['s']`: $[t_0, \ldots, t_{N-1}]$, a list with the dimensions of the positive semidefinite cones (nonnegative integers).

The default value of `dims` is `{'l': G.size[0], 'q': [], 's': []}`, i.e., by default the inequality is interpreted as a componentwise vector inequality.

The arguments `c`, `h` and `b` are real single-column dense matrices. `G` and `A` are real dense or sparse matrices. The number of rows of `G` and `h` is equal to

$$K = l + \sum_{k=0}^{M-1} r_k + \sum_{k=0}^{N-1} t_k^2.$$

The columns of `G` and `h` are vectors in

$$\mathbf{R}^l \times \mathbf{R}^{r_0} \times \cdots \times \mathbf{R}^{r_{M-1}} \times \mathbf{R}^{t_0^2} \times \cdots \times \mathbf{R}^{t_{N-1}^2},$$

where the last N components represent symmetric matrices stored in column major order. The strictly upper triangular entries of these matrices are not accessed (i.e., the symmetric matrices are stored in the 'L'-type column major order used in the `blas` and `lapack` modules). The default values for `A` and `b` are matrices with zero rows, meaning that there are no equality constraints.

`primalstart` is a dictionary with keys `'x'` and `'s'`, used as an optional primal starting point. `primalstart['x']` and `primalstart['s']` are real dense matrices of size $(n, 1)$ and $(K, 1)$, respectively, where n is the length of `c`. The vector `primalstart['s']` must be strictly positive with respect to the cone C.

`dualstart` is a dictionary with keys `'y'` and `'z'`, used as an optional dual starting point. `dualstart['y']` and `dualstart['z']` are real dense matrices of size $(p, 1)$ and $(K, 1)$, respectively, where p is the number of rows in `A`. The vector `dualstart['s']` must be strictly positive with respect to the cone C.

The role of the optional argument `kktsolver` is explained in section 8.7.

`conelp()` returns a dictionary with keys `'status'`, `'x'`, `'s'`, `'y'`, `'z'`. The `'status'` field is a string with possible values `'optimal'`, `'primal infeasible'`, `'dual infeasible'` and `'unknown'`. The meaning of the other fields depends on the value of `'status'`.

`'optimal'` In this case the `'x'`, `'s'`, `'y'` and `'z'` entries contain the primal and dual solutions, which approximately satisfy

$$Gx + s = h, \qquad Ax = b, \qquad G^T z + A^T y + c = 0, \qquad s \succeq 0, \qquad z \succeq 0, \qquad s^T z = 0.$$

`'primal infeasible'` The `'x'` and `'s'` entries are `None`, and the `'y'`, `'z'` entries provide an approximate certificate of infeasibility, i.e., vectors that approximately satisfy

$$G^T z + A^T y = 0, \qquad h^T z + b^T y = -1, \qquad z \succeq 0.$$

`'dual infeasible'` The `'y'` and `'z'` entries are `None`, and the `'x'` and `'s'` entries contain an approximate certificate of dual infeasibility

$$Gx + s = 0, \qquad Ax = 0, \qquad c^T x = -1, \qquad s \succeq 0.$$

`'unknown'` The `'x'`, `'s'`, `'y'`, `'z'` entries are `None`.

It is required that

$$\mathrm{rank}(A) = p, \qquad \mathrm{rank}(\begin{bmatrix} G \\ A \end{bmatrix}) = n,$$

where p is the number or rows of A and n is the number of columns of G and A.

As an example we solve the problem

$$
\begin{array}{ll}
\text{minimize} & -6x_1 - 4x_2 - 5x_3 \\
\text{subject to} & 16x_1 - 14x_2 + 5x_3 \leq -3 \\
& 7x_1 + 2x_2 \leq 5 \\
& \left((8x_1 + 13x_2 - 12x_3 - 2)^2 + (-8x_1 + 18x_2 + 6x_3 - 14)^2 + (x_1 - 3x_2 - 17x_3 - 13)^2\right)^{1/2} \leq -24x_1 - 7x_2 + 15x_3 + \\
& (x_1^2 + x_2^2 + x_3^2)^{1/2} \leq 10 \\
& \begin{bmatrix} 7x_1 + 3x_2 + 9x_3 & -5x_1 + 13x_2 + 6x_3 & x_1 - 6x_2 - 6x_3 \\ -5x_1 + 13x_2 + 6x_3 & x_1 + 12x_2 - 7x_3 & -7x_1 - 10x_2 - 7x_3 \\ x_1 - 6x_2 - 6x_3 & -7x_1 - 10x_2 - 7x_3 & -4x_1 - 28x_2 - 11x_3 \end{bmatrix} \preceq \begin{bmatrix} 68 & -30 & -19 \\ -30 & 99 & 23 \\ -19 & 23 & 10 \end{bmatrix}.
\end{array}
$$

```
>>> from cvxopt.base import matrix
>>> from cvxopt import solvers
>>> c = matrix([-6., -4., -5.])
>>> G = matrix([[ 16., 7.,  24.,  -8.,   8.,  -1.,  0., -1.,  0.,  0.,   7.,  -5.,   1.,  -
                [-14., 2.,   7., -13., -18.,   3.,  0.,  0., -1.,  0.,   3.,  13.,  -6.,  1
                [  5., 0., -15.,  12.,  -6.,  17.,  0.,  0.,  0., -1.,   9.,   6.,  -6.,
>>> h = matrix( [ -3., 5.,  12.,  -2., -14., -13., 10.,  0.,  0.,  0.,  68., -30., -19., -3
>>> dims = {'l': 2, 'q': [4, 4], 's': [3]}
>>> sol = solvers.conelp(c, G, h, dims)
>>> sol['status']
'optimal'
>>> print sol['x']
[-1.22e+00]
[ 9.66e-02]
[ 3.58e+00]
>>> print sol['z']
[ 9.30e-02]
[ 2.04e-08]
[ 2.35e-01]
[ 1.33e-01]
[-4.74e-02]
[ 1.88e-01]
[ 2.79e-08]
[ 1.85e-09]
[-6.32e-10]
[-7.59e-09]
[ 1.26e-01]
[ 8.78e-02]
[-8.67e-02]
[ 8.78e-02]
[ 6.13e-02]
[-6.06e-02]
[-8.67e-02]
[-6.06e-02]
[ 5.98e-02]
```

Only the entries of `G` and `h` defining the lower triangular portions of the coefficients in the linear matrix inequalities are accessed. We obtain the same result if we define `G` and `h` as below.

```
>>> G = matrix([[ 16., 7.,  24.,  -8.,   8.,  -1.,  0., -1.,  0.,  0.,   7.,  -5.,   1.,  0
```

```
                [-14.,  2.,   7., -13., -18.,   3.,  0.,  0., -1.,  0.,   3.,  13.,  -6.,  0
                [  5.,  0., -15.,  12.,  -6.,  17.,  0.,  0.,  0., -1.,   9.,   6.,  -6.,  0
>>> h = matrix( [ -3.,  5.,  12.,  -2., -14., -13., 10.,  0.,  0.,  0.,  68., -30., -19.,  0
```

8.2 Quadratic Cone Programs

coneqp(P, q[, G, h[, dims[, A, b[, initvals[, kktsolver]]]]])

Solves a pair of primal and dual quadratic cone programs

minimize	$(1/2)x^TPx + q^Tx$	maximize	$-(1/2)(q+G^Tz+A^Ty)^TP^\dagger(q+G^Tz+A^Ty) - h^Tz - b^$
subject to	$Gx + s = h$	subject to	$q + G^Tz + A^Ty \in \text{range}(P)$
	$Ax = b$		$z \succeq 0.$
	$s \succeq 0$		

The primal variables are x and the slack variable s. The dual variables are y and z. The inequalities are interpreted as $s \in C$, $z \in C$, where C is a cone defined as a Cartesian product of a nonnegative orthant, a number of second-order cones, and a number of positive semidefinite cones:

$$C = C_0 \times C_1 \times \cdots \times C_M \times C_{M+1} \times \cdots \times C_{M+N}$$

with

$$C_0 = \{u \in \mathbf{R}^l \mid u_k \geq 0,\ k = 1, \ldots, l\}, \qquad C_{k+1} = \{(u_0, u_1) \in \mathbf{R} \times \mathbf{R}^{r_k - 1} \mid u_0 \geq \|u_1\|_2\}, \quad k = 0, \ldots, M-1, \qquad C_{k+M+1}$$

In this definition, $\mathbf{vec}(u)$ denotes a symmetric matrix u stored as a vector in column major order. The structure of C is specified by dims. This argument is a dictionary with three fields.

dims['l']: l, the dimension of the nonnegative orthant (a nonnegative integer).

dims['q']: $[r_0, \ldots, r_{M-1}]$, a list with the dimensions of the second-order cones (positive integers).

dims['s']: $[t_0, \ldots, t_{N-1}]$, a list with the dimensions of the positive semidefinite cones (nonnegative integers).

The default value of dims is {'l': G.size[0], 'q': [], 's': []}, i.e., by default the inequality is interpreted as a componentwise vector inequality.

P is a square dense or sparse real matrix, representing a positive semidefinite symmetric matrix in 'L' storage, i.e., only the lower triangular part of P is referenced. q is a real single-column dense matrix.

The arguments h and b are real single-column dense matrices. G and A are real dense or sparse matrices. The number of rows of G and h is equal to

$$K = l + \sum_{k=0}^{M-1} r_k + \sum_{k=0}^{N-1} t_k^2.$$

The columns of G and h are vectors in

$$\mathbf{R}^l \times \mathbf{R}^{r_0} \times \cdots \times \mathbf{R}^{r_{M-1}} \times \mathbf{R}^{t_0^2} \times \cdots \times \mathbf{R}^{t_{N-1}^2},$$

where the last N components represent symmetric matrices stored in column major order. The strictly upper triangular entries of these matrices are not accessed (i.e., the symmetric matrices are stored in the 'L'-type column major order used in the blas and lapack modules). The default values for G, h, A and b are matrices with zero rows, meaning that there are no inequality or equality constraints.

initvals is a dictionary with keys 'x', 's', 'y', 'z' used as an optional starting point. The vectors initvals['s'] and initvals['z'] must be strictly positive with respect to the cone C. If the argument

`initvals` or any the four entries in it are missing, default starting points are used for the corresponding variables.

The role of the optional argument `kktsolver` is explained in section 8.7.

`coneqp()` returns a dictionary with keys `'status'`, `'x'`, `'s'`, `'y'`, `'z'`. The `'status'` field is a string with possible values `'optimal'` and `'unknown'`.

`'optimal'` In this case the `'x'`, `'s'`, `'y'` and `'z'` entries contain primal and dual solutions, which approximately satisfy

$$Gx + s = h, \qquad Ax = b, \qquad Px + G^T z + A^T y + c = 0, \qquad s \succeq 0, \qquad z \succeq 0, \qquad s^T z = 0.$$

`'unknown'` The `'x'`, `'s'`, `'y'`, `'z'` entries are `None`.

It is required that the problem is solvable and that

$$\text{rank}(A) = p, \qquad \text{rank}(\begin{bmatrix} P \\ G \\ A \end{bmatrix}) = n,$$

where p is the number or rows of A and n is the number of columns of G and A.

As an example, we solve a constrained least-squares problem

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 \\ \text{subject to} & x \succeq 0 \\ & \|x\|_2 \leq 1 \end{array}$$

with

$$A = \begin{bmatrix} 0.3 & 0.6 & -0.3 \\ -0.4 & 1.2 & 0.0 \\ -0.2 & -1.7 & 0.6 \\ -0.4 & 0.3 & -1.2 \\ 1.3 & -0.3 & -2.0 \end{bmatrix}, \qquad b = \begin{bmatrix} 1.5 \\ 0.0 \\ -1.2 \\ -0.7 \\ 0.0 \end{bmatrix}.$$

```
>>> from cvxopt import base, solvers
>>> from cvxopt.base import matrix
>>> A = matrix([ [ .3, -.4,  -.2,  -.4,  1.3 ],
                 [ .6, 1.2, -1.7,   .3,  -.3 ],
                 [-.3,  .0,   .6, -1.2, -2.0 ] ])
>>> b = matrix([ 1.5, .0, -1.2, -.7, .0])
>>> m, n = A.size
>>> I = matrix(0.0, (n,n))
>>> I[::n+1] = 1.0
>>> G = matrix([-I, matrix(0.0, (1,n)), I])
>>> h = matrix(n*[0.0] + [1.0] + n*[0.0])
>>> dims = {'l': n, 'q': [n+1], 's': []}
>>> x = solvers.coneqp(A.T*A, -A.T*b, G, h, dims)['x']
>>> print x
[ 7.26e-01]
[ 6.18e-01]
[ 3.03e-01]
```

8.3 Linear Programming

The function `lp()` is an interface to `conelp()` for linear programs. It also provides the option of using the linear programming solvers from GLPK or MOSEK.

lp(`c, G, h[, A, b[, solver[, primalstart[, dualstart]]]]`)

Solves the pair of primal and dual linear programs

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Gx + s = h \\ & Ax = b \\ & s \succeq 0 \end{array} \qquad \begin{array}{ll} \text{maximize} & -h^T z - b^T y \\ \text{subject to} & G^T z + A^T y + c = 0 \\ & z \succeq 0. \end{array}$$

The inequalities are componentwise vector inequalities.

The `solver` argument is used to choose among three solvers. When it is omitted or `None`, the CVXOPT function `conelp()` is used. The external solvers GLPK and MOSEK (if installed) can be selected by setting `solver = 'glpk'` or `solver = 'mosek'`; see section 8.8.

The meaning of the other arguments and the return value are the same as for `conelp()` called with `dims = {'l': G.size[0], 'q': [], 's': []}`. The initial values are ignored when `solver = 'mosek'` or `solver = 'glpk'`.

As a simple example we solve the LP

$$\begin{array}{ll} \text{minimize} & -4x_1 - 5x_2 \\ \text{subject to} & 2x_1 + x_2 \leq 3 \\ & x_1 + 2x_2 \leq 3 \\ & x_1 \geq 0, \quad x_2 \geq 0. \end{array}$$

```
>>> from cvxopt.base import matrix
>>> from cvxopt import solvers
>>> c = matrix([-4., -5.])
>>> G = matrix([[2., 1., -1., 0.], [1., 2., 0., -1.]])
>>> h = matrix([3., 3., 0., 0.])
>>> sol = solvers.lp(c, G, h)
>>> print sol['x']
[ 1.00e+00]
[ 1.00e+00]
```

8.4 Quadratic Programming

The function `qp()` is an interface to `coneqp()` for quadratic programs. It also provides the option of using the quadratic programming solver from MOSEK.

qp(`P, q[, G, h [, A, b[, solver[, initvals]]]]`)

Solves the pair of primal and dual convex quadratic programs

$$\begin{array}{ll} \text{minimize} & (1/2)x^T P x + q^T x \\ \text{subject to} & Gx \preceq h \\ & Ax = b. \end{array} \qquad \begin{array}{ll} \text{maximize} & -(1/2)(q + G^T z + A^T y)^T P^\dagger (q + G^T z + A^T y) - h^T z - b \\ \text{subject to} & q + G^T z + A^T y \in \text{range}(P) \\ & z \succeq 0. \end{array}$$

The inequalities are componentwise vector inequalities.

The default CVXOPT solver is used when the `solver` argument is absent or `None`. The MOSEK solver (if installed) can be selected by setting `solver = 'mosek'`; see section 8.8. The meaning of the other

arguments and the return value is the same as for `coneqp()` called with `dims = {'l': G.size[0], 'q': [], 's': []}`.

When the `solver = 'mosek'` is used the initial values are ignored, and the `'status'` string in the solution dictionary can take four possible values: `'optimal'`, `'unknown'`. `'primal infeasible'`, `'dual infeasible'`.

`'primal infeasible'` This means that a certificate of primal infeasibility has been found. The `'x'` and `'s'` entries are `None`, and the `'z'` and `'y'` entries are vectors that approximately satisfy

$$G^T z + A^T y = 0, \qquad h^T z + b^T y = -1, \qquad z \succeq 0.$$

`'dual infeasible'` This means that a certificate of dual infeasibility has been found. The `'z'` and `'y'` entries are `None`, and the `'x'` and `'s'` entries are vectors that approximately satisfy

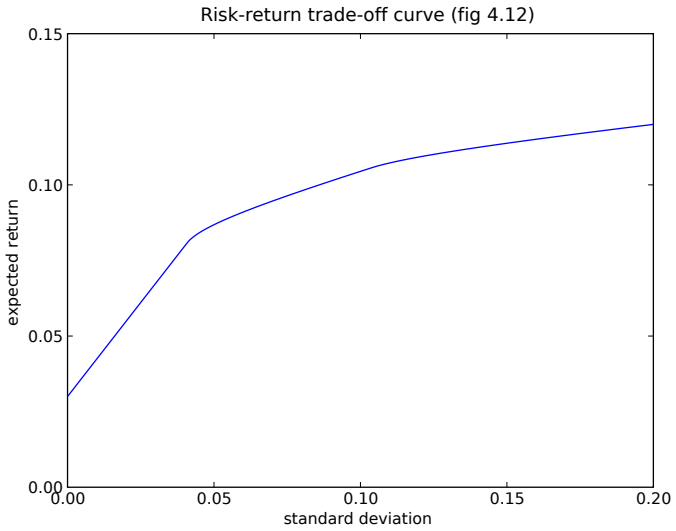
$$Px = 0, \qquad q^T x = -1, \qquad Gx + s = 0, \qquad Ax = 0, \qquad s \succeq 0.$$

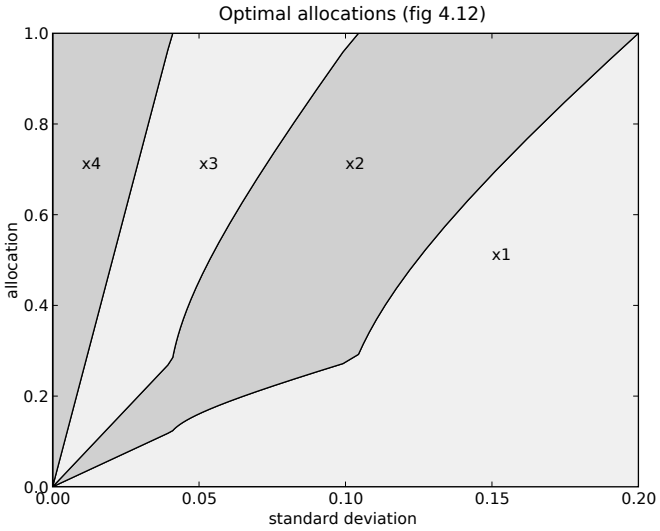
As an example we compute the trade-off curve on page 187 of the book Convex Optimization[1], by solving the quadratic program

$$\begin{array}{ll} \text{minimize} & -\bar{p}^T x + \mu x^T S x \\ \text{subject to} & \mathbf{1}^T x = 1, \quad x \succeq 0 \end{array}$$

for a sequence of positive values of *mu*. The code below computes the trade-off curve and produces two figures using the Matplotlib[2] package.

[1] `http://www.stanford.edu/~{}boyd/cvxbook`
[2] `http://matplotlib.sourceforge.net`






```
from math import sqrt
from cvxopt.base import matrix
from cvxopt.blas import dot
from cvxopt.solvers import qp
import pylab

# Problem data.
n = 4
S = matrix([[ 4e-2,  6e-3, -4e-3,    0.0 ],
            [ 6e-3,  1e-2,  0.0,     0.0 ],
            [-4e-3,  0.0,   2.5e-3,  0.0 ],
            [ 0.0,   0.0,   0.0,     0.0 ]])
pbar = matrix([.12, .10, .07, .03])
G = matrix(0.0, (n,n))
G[::n+1] = -1.0
h = matrix(0.0, (n,1))
```

```
A = matrix(1.0, (1,n))
b = matrix(1.0)

# Compute trade-off.
N = 100
mus = [ 10**(5.0*t/N-1.0) for t in xrange(N) ]
portfolios = [ qp(mu*S, -pbar, G, h, A, b)['x'] for mu in mus ]
returns = [ dot(pbar,x) for x in portfolios ]
risks = [ sqrt(dot(x, S*x)) for x in portfolios ]

# Plot trade-off curve and optimal allocations.
pylab.figure(1, facecolor='w')
pylab.plot(risks, returns)
pylab.xlabel('standard deviation')
pylab.ylabel('expected return')
pylab.axis([0, 0.2, 0, 0.15])
pylab.title('Risk-return trade-off curve (fig 4.12)')
pylab.yticks([0.00, 0.05, 0.10, 0.15])

pylab.figure(2, facecolor='w')
c1 = [ x[0] for x in portfolios ]
c2 = [ x[0] + x[1] for x in portfolios ]
c3 = [ x[0] + x[1] + x[2] for x in portfolios ]
c4 = [ x[0] + x[1] + x[2] + x[3] for x in portfolios ]
pylab.fill(risks + [.20], c1 + [0.0], '#F0F0F0')
pylab.fill(risks[-1::-1] + risks, c2[-1::-1] + c1, facecolor = '#D0D0D0')
pylab.fill(risks[-1::-1] + risks, c3[-1::-1] + c2, facecolor = '#F0F0F0')
pylab.fill(risks[-1::-1] + risks, c4[-1::-1] + c3, facecolor = '#D0D0D0')
pylab.axis([0.0, 0.2, 0.0, 1.0])
pylab.xlabel('standard deviation')
pylab.ylabel('allocation')
pylab.text(.15,.5,'x1')
pylab.text(.10,.7,'x2')
pylab.text(.05,.7,'x3')
pylab.text(.01,.7,'x4')
pylab.title('Optimal allocations (fig 4.12)')
pylab.show()
```

8.5 Second-Order Cone Programming

The function `socp()` is a simpler interface to `conelp()` for cone programs with no linear matrix inequality constraints.

socp(`c[, Gl, hl[, Gq, hq[, A, b[, solver[, primalstart[, dualstart]]]]]]`)

Solves the pair of primal and dual second-order cone programs

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & G_k x + s_k = h_k, \quad k = 0, \ldots, M \\ & Ax = b \\ & s_0 \succeq 0 \\ & s_{k0} \geq \|s_{k1}\|_2, \quad k = 1, \ldots, M \end{array}$$

$$\begin{array}{ll} \text{maximize} & -\sum_{k=0}^{M} h_k^T z_k - b^T y \\ \text{subject to} & \sum_{k=0}^{M} G_k^T z_k + A^T y + c = 0 \\ & z_0 \succeq 0 \\ & z_{k0} \geq \|z_{k1}\|_2, \quad k = 1, \ldots, M. \end{array}$$

The inequalities

$$s_0 \succeq 0, \qquad z_0 \succeq 0$$

are componentwise vector inequalities. In the other inequalities, it is assumed that the variables are partitioned as

$$s_k = (s_{k0}, s_{k1}) \in \mathbf{R} \times \mathbf{R}^{r_k - 1}, \qquad z_k = (z_{k0}, z_{k1}) \in \mathbf{R} \times \mathbf{R}^{r_k - 1}, \qquad k = 1, \ldots, M.$$

The input argument `c` is a real single-column dense matrix. The arguments `Gl` and `hl` are the coefficient matrix G_0 and the righthand side h_0 of the componentwise inequalities. `Gl` is a real dense or sparse matrix; `hl` is a real single-column dense matrix. The default values for `Gl` and `hl` are matrices with zero rows.

The argument `Gq` is a list of M dense or sparse matrices $G_1, \ldots, G_M$. The argument `hq` is a list of M dense single-column matrices $h_1, \ldots, h_M$. The elements of `Gq` and `hq` must have at least one row. The default values of `Gq` and `hq` are empty lists.

`A` is dense or sparse matrix and `b` is a single-column dense matrix. The default values for `A` and `b` are matrices with zero rows.

The `solver` argument is used to choose between two solvers: the CVXOPT `conelp()` solver (used when `solver` is absent or equal to `None`) and the external solver MOSEK (`solver='mosek'`); see section 8.8. With the `'mosek'` option the code does not accept problems with equality constraints.

`primalstart` and `dualstart` are dictionaries with optional primal, respectively, dual starting points. `primalstart` has elements `'x'`, `'sl'`, `'sq'`. `primalstart['x']` and `primalstart['sl']` are single-column dense matrices with the initial values of x and s_0; `primalstart['sq']` is a list of single-column matrices with the initial values of $s_1, \ldots, s_M$. The initial values must satisfy the inequalities in the primal problem strictly, but not necessarily the equality constraints.

`dualstart` has elements `'y'`, `'zl'`, `'zq'`. `dualstart['y']` and `dualstart['zl']` are single-column dense matrices with the initial values of y and z_0. `dualstart['zq']` is a list of single-column matrices with the initial values of $z_1, \ldots, z_M$. These values must satisfy the dual inequalities strictly, but not necessarily the equality constraint.

The arguments `primalstart` and `dualstart` are ignored when the MOSEK solver is used.

`socp()` returns a dictionary with keys `'status'`, `'x'`, `'sl'`, `'sq'`, `'y'`, `'zl'`, `'zq'`. The meaning is similar to the output of `conelp()`. The `'sl'` and `'zl'` fields are matrices with the primal slacks and dual variables associated with the componentwise linear inequalities. The `'sq'` and `'zq'` fields are lists with the primal slacks and dual variables associated with the second-order cone inequalities.

As an example, we solve the second-order cone program

$$\begin{array}{ll} \text{minimize} & -2x_1 + x_2 + 5x_3 \\ \text{subject to} & \left\| \begin{bmatrix} -13x_1 + 3x_2 + 5x_3 - 3 \\ -12x_1 + 12x_2 - 6x_3 - 2 \end{bmatrix} \right\|_2 \leq -12x_1 - 6x_2 + 5x_3 - 12 \\ & \left\| \begin{bmatrix} -3x_1 + 6x_2 + 2x_3 \\ x_1 + 9x_2 + 2x_3 + 3 \\ -x_1 - 19x_2 + 3x_3 - 42 \end{bmatrix} \right\|_2 \leq -3x_1 + 6x_2 - 10x_3 + 27. \end{array}$$

```
>>> from cvxopt.base import matrix
>>> from cvxopt import solvers
>>> c = matrix([-2., 1., 5.])
>>> G = [ matrix( [[12., 13., 12.], [6., -3., -12.], [-5., -5., 6.]] ) ]
>>> G += [ matrix( [[3., 3., -1., 1.], [-6., -6., -9., 19.], [10., -2., -2., -3.]] ) ]
>>> h = [ matrix( [-12., -3., -2.] ),  matrix( [27., 0., 3., -42.] ) ]
>>> sol = solvers.socp(c, Gq = G, hq = h)
>>> sol['status']
optimal
```

```
>>> print sol['x']
[-5.02e+00]
[-5.77e+00]
[-8.52e+00]
>>> print sol['zq'][0]
[ 1.34e+00]
[-7.63e-02]
[-1.34e+00]
>>> print sol['zq'][1]
[ 1.02e+00]
[ 4.02e-01]
[ 7.80e-01]
[-5.17e-01]
```

8.6 Semidefinite Programming

The function `sdp()` is a simple interface to `conelp()` for cone programs with no second-order cone constraints. It also provides the option of using the DSDP semidefinite programming solver.

sdp(`c[, Gl, hl[, Gs, hs[, A, b[, solver[, primalstart[, dualstart]]]]]]`)

Solves the pair of primal and dual semidefinite programs

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & G_0 x + s_0 = h_0 \\ & G_k x + \mathbf{vec}\,(s_k) = \mathbf{vec}\,(h_k), \quad k = 1, \ldots, N \\ & Ax = b \\ & s_0 \succeq 0 \\ & s_k \succeq 0, \quad k = 1, \ldots, N \end{array}$$

$$\begin{array}{ll} \text{maximize} & -h_0^T z_0 - \sum_{k=1}^N \mathbf{tr}(h_k z_k) - b^T y \\ \text{subject to} & G_0^T z_0 + \sum_{k=1}^N G_k^T\, \mathbf{vec}(z_k) + A^T y + c = 0 \\ & z_0 \succeq 0 \\ & z_k \succeq 0, \quad k = 1, \ldots, N. \end{array}$$

The inequalities

$$s_0 \succeq 0, \qquad z_0 \succeq 0$$

are componentwise vector inequalities. The other inequalities are matrix inequalities (i.e., the require the left-hand sides to be positive semidefinite). We use the notation $\mathbf{vec}(z)$ to denote a symmetric matrix z stored in column major order as a column vector.

The input argument `c` is a real single-column dense matrix. The arguments `Gl` and `hl` are the coefficient matrix G_0 and the righthand side h_0 of the componentwise inequalities. `Gl` is a real dense or sparse matrix; `hl` is a real single-column dense matrix. The default values for `Gl` and `hl` are matrices with zero rows.

`Gs` and `hs` are lists of length N that specify the linear matrix inequality constraints. `Gs` is a list of N dense or sparse real matrices $G_1, \ldots, G_M$. The columns of these matrices can be interpreted as symmetric matrices stored in column major order, using the BLAS 'L'-type storage (i.e., only the entries corresponding to lower triangular positions are accessed). `hs` is a list of N dense symmetric matrices $h_1, \ldots, h_N$. Only the lower triangular elements of these matrices are accessed. The default values for `Gs` and `hs` are empty lists.

`A` is a dense or sparse matrix and `b` is a single-column dense matrix. The default values for `A` and `b` are matrices with zero rows.

The `solver` argument is used to choose between two solvers: the CVXOPT `conelp()` solver (used when `solver` is absent or equal to `None`) and the external solver DSDP5 (`solver='dsdp'`); see section 8.8. With the `'dsdp'` option the code does not accept problems with equality constraints.

The optional argument `primalstart` is a dictionary with keys `'x'`, `'sl'`, and `'ss'`, used as an optional primal starting point. `primalstart['x']` and `primalstart['sl']` are single-column dense matrices with the initial values of x and s_0; `primalstart['ss']` is a list of square matrices with the initial values of

$s_1, \ldots, s_N$. The initial values must satisfy the inequalities in the primal problem strictly, but not necessarily the equality constraints.

`dualstart` is a dictionary with keys `'y'`, `'zl'`, `'zs'`, used as an optional dual starting point. `dualstart['y']` and `dualstart['zl']` are single-column dense matrices with the initial values of y and z_0. `dualstart['zs']` is a list of square matrices with the initial values of $z_1, \ldots, z_N$. These values must satisfy the dual inequalities strictly, but not necessarily the equality constraint.

The arguments `primalstart` and `dualstart` are ignored when the DSDP solver is used.

`sdp()` returns a dictionary with keys `'status'`, `'x'`, `'sl'`, `'ss'`, `'y'`, `'zl'`, `'ss'`. The `'sl'` and `'zl'` fields are matrices with the primal slacks and dual variables associated with the componentwise linear inequalities. The `'ss'` and `'zs'` fields are lists with the primal slacks and dual variables associated with the second-order cone inequalities.

We illustrate the calling sequence with a small example.

$$
\begin{array}{ll}
\text{minimize} & x_1 - x_2 + x_3 \\
\text{subject to} & x_1 \begin{bmatrix} -7 & -11 \\ -11 & 3 \end{bmatrix} + x_2 \begin{bmatrix} 7 & -18 \\ -18 & 8 \end{bmatrix} + x_3 \begin{bmatrix} -2 & -8 \\ -8 & 1 \end{bmatrix} \preceq \begin{bmatrix} 33 & -9 \\ -9 & 26 \end{bmatrix} \\
& x_1 \begin{bmatrix} -21 & -11 & 0 \\ -11 & 10 & 8 \\ 0 & 8 & 5 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 10 & 16 \\ 10 & -10 & -10 \\ 16 & -10 & 3 \end{bmatrix} + x_3 \begin{bmatrix} -5 & 2 & -17 \\ 2 & -6 & -7 \\ -17 & 8 & 6 \end{bmatrix} \preceq \begin{bmatrix} 14 & 9 & 40 \\ 9 & 91 & 10 \\ 40 & 10 & 15 \end{bmatrix}
\end{array}
$$

```
>>> from cvxopt.base import matrix
>>> from cvxopt import solvers
>>> c = matrix([1.,-1.,1.])
>>> G = [ matrix([[-7., -11., -11., 3.],
                  [ 7., -18., -18., 8.],
                  [-2.,  -8.,  -8., 1.]]) ]
>>> G += [ matrix([[-21., -11.,   0., -11.,  10.,   8.,   0.,   8., 5.],
                   [  0.,  10.,  16.,  10., -10., -10.,  16., -10., 3.],
                   [ -5.,   2., -17.,   2.,  -6.,   8., -17.,  -7., 6.]]) ]
>>> h = [ matrix([[33., -9.], [-9., 26.]]) ]
>>> h += [ matrix([[14., 9., 40.], [9., 91., 10.], [40., 10., 15.]]) ]
>>> sol = solvers.sdp(c, Gs=G, hs=h)
>>> print sol['x']
[-3.68e-01]
[ 1.90e+00]
[-8.88e-01]
>>> print sol['zs'][0]
[ 3.96e-03 -4.34e-03]
[-4.34e-03  4.75e-03]
>>> print sol['zs'][1]
[ 5.58e-02 -2.41e-03  2.42e-02]
[-2.41e-03  1.04e-04 -1.05e-03]
[ 2.42e-02 -1.05e-03  1.05e-02]
```

Only the entries in `Gs` and `hs` that correspond to lower triangular entries need to be provided, so in the example `h` and `G` may also be defined as follows.

```
>>> G = [ matrix([[-7., -11., 0., 3.],
                  [ 7., -18., 0., 8.],
                  [-2.,  -8., 0., 1.]]) ]
>>> G += [ matrix([[-21., -11.,   0., 0.,  10.,   8., 0., 0., 5.],
                   [  0.,  10.,  16., 0., -10., -10., 0., 0., 3.],
```

```
                    [ -5.,   2., -17., 0.,  -6.,   8., 0., 0., 6.]]) ]
>>> h = [ matrix([[33., -9.], [0., 26.]]) ]
>>> h += [ matrix([[14., 9., 40.], [0., 91., 10.], [0., 0., 15.]]) ]
```

8.7 Exploiting Structure

By default, the functions `conelp()` and `coneqp()` exploit no problem structure except (to some limited extent) sparsity. Two mechanisms are provided for implementing customized solvers that take advantage of problem structure.

Providing a function for solving KKT equations. The most expensive step of each iteration of `conelp()` or `coneqp()` is the solution of a set of linear equations ('KKT equations') of the form

$$\begin{bmatrix} P & A^T & G^T \\ A & 0 & 0 \\ G & 0 & -W^TW \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} \tag{8.1}$$

(with $P = 0$ in `conelp()`). The matrix W depends on the current iterates and is defined as follows. We use the notation of sections 8.1 and 8.2. Suppose

$$u = (u_{\mathrm{l}},\ u_{\mathrm{q},0},\ \ldots,\ u_{\mathrm{q},M-1},\ \mathbf{vec}\,(u_{\mathrm{s},0}),\ \ldots,\ \mathbf{vec}\,(u_{\mathrm{s},N-1}))\,, \qquad u_{\mathrm{l}} \in \mathbf{R}^l, \qquad u_{\mathrm{q},k} \in \mathbf{R}^{r_k}, \quad k = 0, \ldots, M-1, \qquad u_{\mathrm{s},k} \in \mathbf{S}$$

Then W is a block-diagonal matrix,

$$Wu = (W_{\mathrm{l}}u_{\mathrm{l}},\ W_{\mathrm{q},0}u_{\mathrm{q},0},\ \ldots,\ W_{\mathrm{q},M-1}u_{\mathrm{q},M-1},\ W_{\mathrm{s},0}\,\mathbf{vec}\,(u_{\mathrm{s},0}),\ \ldots,\ W_{\mathrm{s},N-1}\,\mathbf{vec}\,(u_{\mathrm{s},N-1}))$$

with the following diagonal blocks.

- The first block is a *positive diagonal scaling* with a vector d:

  $$W_{\mathrm{l}} = \mathbf{diag}\,(d), \qquad W_{\mathrm{l}}^{-1} = \mathbf{diag}\,(d)^{-1}.$$

  This transformation is symmetric:

  $$W_{\mathrm{l}}^T = W_{\mathrm{l}}.$$

- The next M blocks are positive multiples of *hyperbolic Householder transformations*:

  $$W_{\mathrm{q},k} = \beta_k(2v_kv_k^T - J), \qquad W_{\mathrm{q},k}^{-1} = \frac{1}{\beta_k}(2Jv_kv_k^TJ - J), \qquad k = 0, \ldots, M-1,$$

  where

  $$\beta_k > 0, \qquad v_{k0} > 0, \qquad v_k^TJv_k = 1, \qquad J = \begin{bmatrix} 1 & 0 \\ 0 & -I \end{bmatrix}.$$

  These transformations are also symmetric:

  $$W_{\mathrm{q},k}^T = W_{\mathrm{q},k}.$$

- The last N blocks are *congruence transformations* with nonsingular matrices:

  $$W_{\mathrm{s},k}\,\mathbf{vec}\,(u_{\mathrm{s},k}) = \mathbf{vec}\,(r_k^Tu_{\mathrm{s},k}r_k), \qquad W_{\mathrm{s},k}^{-1}\,\mathbf{vec}\,(u_{\mathrm{s},k}) = \mathbf{vec}\,(r_k^{-T}u_{\mathrm{s},k}r_k^{-1}), \qquad k = 0, \ldots, N-1.$$

  In general, this operation is not symmetric:

  $$W_{\mathrm{s},k}^T\,\mathbf{vec}\,(u_{\mathrm{s},k}) = \mathbf{vec}\,(r_ku_{\mathrm{s},k}r_k^T), \qquad W_{\mathrm{s},k}^{-T}\,\mathbf{vec}\,(u_{\mathrm{s},k}) = \mathbf{vec}\,(r_k^{-1}u_{\mathrm{s},k}r_k^{-T}), \qquad k = 0, \ldots, N-1.$$

It is often possible to exploit problem structure to solve (8.1) faster than by standard methods. The last argument `kktsolver` of `conelp()` and `coneqp()` allows the user to supply a Python function for solving the KKT equations. This function will be called as "`f = kktsolver(W)`", where `W` is a dictionary that contains the parameters of the scaling:

- `W['d']` is the positive vector that defines the diagonal scaling. `W['di']` is its componentwise inverse.
- `W['beta']` and `W['v']` are lists of length M with the coefficients and vectors that define the hyperbolic Householder transformations.
- `W['r']` is a list of length N with the matrices that define the the congruence transformations. `W['rti']` is a list of length N with the transposes of the inverses of the matrices in `W['r']`.

The function call "`f = kktsolver(W)`" should return a routine for solving the KKT system (8.1) defined by `W`. It will be called as "`f(bx, by, bz)`". On entry, `bx`, `by`, `bz` contain the righthand side. On exit, they should contain the solution of the KKT system, with the last component scaled, i.e., on exit,

$$b_x := u_x, \qquad b_y := u_y, \qquad b_z := Wu_z.$$

In other words, the function returns the solution of

$$\begin{bmatrix} P & A^T & G^TW^{-1} \\ A & 0 & 0 \\ G & 0 & -W^T \end{bmatrix} \begin{bmatrix} \hat{u}_x \\ \hat{u}_y \\ \hat{u}_z \end{bmatrix} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}.$$

Specifying constraints via Python functions. In the default use of `conelp()` and `coneqp()`, the linear constraints and the quadratic term in the objective are parameterized by CVXOPT matrices `G`, `A`, `P`. It is possible to specify these parameters via Python functions that evaluate the corresponding matrix-vector products and their adjoints.

If the argument `G` of `conelp()` or `coneqp()` is a Python function, it should be defined as follows:

`G(x, y [, alpha[, beta[, trans]]])`

This evaluates the matrix-vector products

$$y := \alpha Gx + \beta y \quad (\text{trans} = \text{'N'}), \qquad y := \alpha G^T x + \beta y \quad (\text{trans} = \text{'T'}).$$

The default values of the optional arguments must be `alpha = 1.0`, `beta = 0.0`, `trans = 'N'`.

Similarly, if the argument `A` is a Python function, then it must be defined as follows.

`A(x, y [, alpha[, beta[, trans]]])`

This evaluates the matrix-vector products

$$y := \alpha Ax + \beta y \quad (\text{trans} = \text{'N'}), \qquad y := \alpha A^T x + \beta y \quad (\text{trans} = \text{'T'}).$$

The default values of the optional arguments must be `alpha = 1.0`, `beta = 0.0`, `trans = 'N'`.

If the argument `P` of `coneqp()` is a Python function, then it must be defined as follows:

`P(x, y [, alpha[, beta]])`

This evaluates the matrix-vector products

$$y := \alpha Px + \beta y.$$

The default values of the optional arguments must be `alpha = 1.0`, `beta = 0.0`.

If `G`, `A` or `P` are Python functions, then the argument `kktsolver` must also be provided.

We illustrate these features with three applications.

Example: 1-norm approximation The optimization problem

$$\text{minimize} \quad \|Pu - q\|_1$$

can be formulated as a linear program

$$\begin{array}{ll} \text{minimize} & \mathbf{1}^T v \\ \text{subject to} & -v \preceq Pu - q \preceq v. \end{array}$$

By exploiting the structure in the inequalities, the cost of an iteration of an interior-point method can be reduced to the cost of least-squares problem of the same dimensions. (See section 11.8.2 in the book Convex Optimization[3].) The code belows takes advantage of this fact.

```
from cvxopt import base, blas, lapack, solvers
from cvxopt.base import matrix, spmatrix, mul, div

def l1(P, q):
    """

    Returns the solution u, w of the l1 approximation problem

        (primal) minimize    ||P*u - q||_1

        (dual)   maximize    q'*w
                 subject to  P'*w = 0
                             ||w||_infty <= 1.
    """

    m, n = P.size

    # Solve the equivalent LP
    #
    #     minimize    [0; 1]' * [u; v]
    #     subject to  [P, -I; -P, -I] * [u; v] <= [q; -q]
    #
    #     maximize    -[q; -q]' * z
    #     subject to  [P', -P']*z  = 0
    #                 [-I, -I]*z + 1 = 0
    #                 z >= 0.

    c = matrix(n*[0.0] + m*[1.0])

    def G(x, y, alpha = 1.0, beta = 0.0, trans = 'N'):

        if trans=='N':
            # y := alpha * [P, -I; -P, -I] * x + beta*y
            u = P*x[:n]
            y[:m] = alpha * ( u - x[n:]) + beta * y[:m]
            y[m:] = alpha * (-u - x[n:]) + beta * y[m:]
```

[3] http://www.ee.ucla.edu/~{}vandenbe/cvxbook

```
    else:
        # y := alpha * [P', -P'; -I, -I] * x + beta*y
        y[:n] =  alpha * P.T * (x[:m] - x[m:]) + beta * y[:n]
        y[n:] = -alpha * (x[:m] + x[m:]) + beta * y[n:]

h = matrix([q, -q])
dims = {'l': 2*m, 'q': [], 's': []}

def F(W):

    """
    Returns a function f(x, y, z) that solves

        [ 0  0  P'      -P'      ] [ x[:n] ]   [ bx[:n] ]
        [ 0  0 -I       -I       ] [ x[n:] ]   [ bx[n:] ]
        [ P -I -D1^{-1}  0       ] [ z[:m] ] = [ bz[:m] ]
        [-P -I  0       -D2^{-1} ] [ z[m:] ]   [ bz[m:] ]

    where D1 = diag(di[:m])^2, D2 = diag(di[m:])^2 and di = W['di'].
    """

    # Factor A = 4*P'*D*P where D = d1.*d2 ./(d1+d2) and d1 = di[:m].^2, d2 = di[m

    di = W['di']
    d1, d2 = di[:m]**2, di[m:]**2
    D = div( mul(d1,d2), d1+d2 )
    A = P.T * spmatrix(4*D, range(m), range(m)) * P
    lapack.potrf(A)

    def f(x, y, z):

        """
        On entry bx, bz are stored in x, z.  On exit x, z contain the solution,
        with z scaled: z./di is returned instead of z.
        """"

        # Solve for x[:n]:
        #
        #    A*x[:n] = bx[:n] + P' * ( ((D1-D2)*(D1+D2)^{-1})*bx[n:]
        #              + (2*D1*D2*(D1+D2)^{-1}) * (bz[:m] - bz[m:]) ).

        x[:n] += P.T * ( mul(div(d1-d2, d1+d2), x[n:]) + mul(2*D, z[:m]-z[m:]) )
        lapack.potrs(A, x)

        # x[n:] := (D1+D2)^{-1} * (bx[n:] - D1*bz[:m] - D2*bz[m:] + (D1-D2)*P*x[:n

        u = P*x[:n]
        x[n:] =  div(x[n:] - mul(d1, z[:m]) - mul(d2, z[m:]) + mul(d1-d2, u), d1+d

        # z[:m] := d1[:m] .* ( P*x[:n] - x[n:] - bz[:m])
        # z[m:] := d2[m:] .* (-P*x[:n] - x[n:] - bz[m:])
```

```
            z[:m] = mul(d[:m],  u - x[n:] - z[:m])
            z[m:] = mul(d[m:], -u - x[n:] - z[m:])

        return f

    sol = solvers.conelp(c, G, h, dims, kktsolver = F)
    return sol['x'][:n],  sol['z'][m:] - sol['z'][:m]
```

Example: SDP with diagonal linear term The SDP

$$\begin{array}{ll} \text{minimize} & \mathbf{1}^T x \\ \text{subject to} & W + \mathbf{diag}\,(x) \succeq 0 \end{array}$$

can be solved efficiently by exploiting properties of the diag operator.

```
from cvxopt import base, blas, lapack, solvers
from cvxopt.base import matrix

def mcsdp(w):
    """
    Returns solution x, z to

        (primal)  minimize    sum(x)
                  subject to  w + diag(x) >= 0

        (dual)    maximize    -tr(w*z)
                  subject to  diag(z) = 1
                              z >= 0.
    """

    n = w.size[0]
    c = matrix(1.0, (n,1))

    def G(x, y, alpha = 1.0, beta = 0.0, trans = 'N'):
        """
            y := alpha*(-diag(x)) + beta*y.
        """

        if trans=='N':
            # x is a vector; y is a symmetric matrix in column major order.
            y *= beta
            y[::n+1] -= alpha * x

        else:
            # x is a symmetric matrix in column major order; y is a vector.
            y *= beta
            y -= alpha * x[::n+1]


    def cngrnc(r, x, alpha = 1.0):
        """
        Congruence transformation
```

```
        x := alpha * r'*x*r.

    r and x are square matrices.
    """

    # Scale diagonal of x by 1/2.
    x[::n+1] *= 0.5

    # a := tril(x)*r
    a = +r
    tx = matrix(x, (n,n))
    blas.trmm(tx, a, side = 'L')

    # x := alpha*(a*r' + r*a')
    blas.syr2k(r, a, tx, trans = 'T', alpha = alpha)
    x[:] = tx[:]

dims = {'l': 0, 'q': [], 's': [n]}

def F(W):
    """
    Returns a function f(x, y, z) that solves

                  -diag(z)     = bx
        -diag(x) - r*r'*z*r*r' = bz

    where r = W['r'][0] = W['rti'][0]^{-T}.
    """

    rti = W['rti'][0]

    # t = rti*rti' as a nonsymmetric matrix.
    t = matrix(0.0, (n,n))
    blas.gemm(rti, rti, t, transB = 'T')

    # Cholesky factorization of tsq = t.*t.
    tsq = t**2
    lapack.potrf(tsq)

    def f(x, y, z):
        """
        On entry, x contains bx, y is empty, and z contains bz stored
        in column major order.
        On exit, they contain the solution, with z scaled
        (vec(r'*z*r) is returned instead of z).

        We first solve

            ((rti*rti') .* (rti*rti')) * x = bx - diag(t*bz*t)

        and take z = - rti' * (diag(x) + bz) * rti.
```

```
            """

            # tbst := t * bz * t
            tbst = +z
            cngrnc(t, tbst)

            # x := x - diag(tbst) = bx - diag(rti*rti' * bz * rti*rti')
            x -= tbst[::n+1]

            # x := (t.*t)^{-1} * x = (t.*t)^{-1} * (bx - diag(t*bz*t))
            lapack.potrs(tsq, x)

            # z := z + diag(x) = bz + diag(x)
            z[::n+1] += x

            # z := -vec(rti' * z * rti)
            #    = -vec(rti' * (diag(x) + bz) * rti
            cngrnc(rti, z, alpha = -1.0)

        return f

    sol = solvers.conelp(c, G, w[:], dims, kktsolver = F)
    return sol['x'], sol['z']
```

Example: Minimizing 1-norm subject to a 2-norm constraint In the second example, we use a similar trick to solve the problem

$$\begin{array}{ll} \text{minimize} & \|u\|_1 \\ \text{subject to} & \|Au - b\|_2 \leq 1. \end{array}$$

The code below is efficient, if we assume that the number of rows in A is greater than or equal to the number of columns.

```
def qcl1(A, b):
    """
    Returns the solution u, z of

        (primal)  minimize    || u ||_1
                  subject to  || A * u - b ||_2  <= 1

        (dual)    maximize    b^T z - ||z||_2
                  subject to  || A'*z ||_inf <= 1.

    Exploits structure, assuming A is m by n with m >= n.
    """

    m, n = A.size

    # Solve equivalent cone LP with variables x = [u; v].
    #
    #     minimize    [0; 1]' * x
    #     subject to  [ I  -I ] * x <=  [  0 ]   (componentwise)
    #                 [-I  -I ] * x <=  [  0 ]   (componentwise)
```

```
#                 [ 0   0 ] * x <=  [  1 ]   (SOC)
#                 [-A   0 ]         [ -b ]
#
#     maximize    -t + b' * w
#     subject to  z1 - z2 = A'*w
#                 z1 + z2 = 1
#                 z1 >= 0,  z2 >=0,  ||w||_2 <= t.

c = matrix(n*[0.0] + n*[1.0])
h = matrix( 0.0, (2*n + m + 1, 1))
h[2*n] = 1.0
h[2*n+1:] = -b

def G(x, y, alpha = 1.0, beta = 0.0, trans = 'N'):
    y *= beta
    if trans=='N':
        # y += alpha * G * x
        y[:n] += alpha * (x[:n] - x[n:2*n])
        y[n:2*n] += alpha * (-x[:n] - x[n:2*n])
        y[2*n+1:] -= alpha * A*x[:n]

    else:
        # y += alpha * G'*x
        y[:n] += alpha * (x[:n] - x[n:2*n] - A.T * x[-m:])
        y[n:] -= alpha * (x[:n] + x[n:2*n])


def Fkkt(W):
    """
    Returns a function f(x, y, z) that solves

        [ 0   G'   ] [ x ] = [ bx ]
        [ G  -W'*W ] [ z ]   [ bz ].
    """

    # First factor
    #
    #     S = G' * W**-1 * W**-T * G
    #       = [0; -A]' * W3^-2 * [0; -A] + 4 * (W1**2 + W2**2)**-1
    #
    # where
    #
    #     W1 = diag(d1) with d1 = W['d'][:n] = 1 ./ W['di'][:n]
    #     W2 = diag(d2) with d2 = W['d'][n:] = 1 ./ W['di'][n:]
    #     W3 = beta * (2*v*v' - J),  W3^-1 = 1/beta * (2*J*v*v'*J - J)
    #        with beta = W['beta'][0], v = W['v'][0], J = [1, 0; 0, -I].

    # As = W3^-1 * [ 0 ; -A ] = 1/beta * ( 2*J*v * v' - I ) * [0; A]
    beta, v = W['beta'][0], W['v'][0]
    As = 2 * v * (v[1:].T * A)
    As[1:,:] *= -1.0
    As[1:,:] -= A
```

```
        As /= beta

        # S = As'*As + 4 * (W1**2 + W2**2)**-1
        S = As.T * As
        d1, d2 = W['d'][:n], W['d'][n:]
        d = 4.0 * (d1**2 + d2**2)**-1
        S[::n+1] += d
        lapack.potrf(S)

        def f(x, y, z):

            # z := - W**-T * z
            z[:n] = -div( z[:n], d1 )
            z[n:2*n] = -div( z[n:2*n], d2 )
            z[2*n:] -= 2.0*v*( v[0]*z[2*n] - blas.dot(v[1:], z[2*n+1:]) )
            z[2*n+1:] *= -1.0
            z[2*n:] /= beta

            # x := x - G' * W**-1 * z
            x[:n] -= div(z[:n], d1) - div(z[n:2*n], d2) + As.T * z[-(m+1):]
            x[n:] += div(z[:n], d1) + div(z[n:2*n], d2)

            # Solve for x[:n]:
            #
            #    S*x[:n] = x[:n] - (W1**2 - W2**2)(W1**2 + W2**2)^-1 * x[n:]

            x[:n] -= mul( div(d1**2 - d2**2, d1**2 + d2**2), x[n:])
            lapack.potrs(S, x)

            # Solve for x[n:]:
            #
            #    (d1**-2 + d2**-2) * x[n:] = x[n:] + (d1**-2 - d2**-2)*x[:n]

            x[n:] += mul( d1**-2 - d2**-2, x[:n])
            x[n:] = div( x[n:], d1**-2 + d2**-2)

            # z := z + W^-T * G*x
            z[:n] += div( x[:n] - x[n:2*n], d1)
            z[n:2*n] += div( -x[:n] - x[n:2*n], d2)
            z[2*n:] += As*x[:n]

        return f

    dims = {'l': 2*n, 'q': [m+1], 's': []}
    sol = solvers.conelp(c, G, h, dims, kktsolver = Fkkt)
    if sol['status'] == 'optimal':
        return sol['x'][:n],  sol['z'][-m:]
    else:
        return None, None
```

Example: 1-norm regularized least-squares As an example that illustrates how structure can be exploited in `coneqp()`,

we consider the 1-norm regularized least-squares problem

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

with variable x. The problem is equivalent to the quadratic program

$$\begin{array}{ll} \text{minimize} & \|Ax - y\|_2^2 + \mathbf{1}^T u \\ \text{subject to} & -u \preceq x \preceq u \end{array}$$

with variables x and u. The implementation below is efficient when A has many more columns than rows.

```
from cvxopt.base import matrix, spdiag, mul, div
from cvxopt import base, blas, lapack, solvers
import math

def l1regls(A, y):
    """

    Returns the solution of l1-norm regularized least-squares problem

        minimize || A*x - y ||_2^2  + || x ||_1.

    """

    m, n = A.size
    q = matrix(1.0, (2*n,1))
    q[:n] = -2.0 * A.T * y

    def P(u, v, alpha = 1.0, beta = 0.0 ):
        """
            v := alpha * 2.0 * [ A'*A, 0; 0, 0 ] * u + beta * v
        """
        v *= beta
        v[:n] += alpha * 2.0 * A.T * (A * u[:n])


    def G(u, v, alpha=1.0, beta=0.0, trans='N'):
        """
            v := alpha*[I, -I; -I, -I] * u + beta * v  (trans = 'N' or 'T')
        """

        v *= beta
        v[:n] += alpha*(u[:n] - u[n:])
        v[n:] += alpha*(-u[:n] - u[n:])

    h = matrix(0.0, (2*n,1))


    # Customized solver for the KKT system
    #
    #     [  2.0*A'*A  0    I      -I     ] [x[:n] ]     [bx[:n] ]
    #     [  0         0   -I      -I     ] [x[n:] ]  =  [bx[n:] ].
    #     [  I        -I   -D1^-1   0     ] [zl[:n]]     [bzl[:n]]
```

```
#     [ -I        -I    0      -D2^-1 ] [zl[n:]]     [bzl[n:]]
#
# where D1 = W['di'][:n]**2, D2 = W['di'][:n]**2.
#
# We first eliminate zl and x[n:]:
#
#     ( 2*A'*A + 4*D1*D2*(D1+D2)^-1 ) * x[:n] =
#         bx[:n] - (D2-D1)*(D1+D2)^-1 * bx[n:] +
#         D1 * ( I + (D2-D1)*(D1+D2)^-1 ) * bzl[:n] -
#         D2 * ( I - (D2-D1)*(D1+D2)^-1 ) * bzl[n:]
#
#     x[n:] = (D1+D2)^-1 * ( bx[n:] - D1*bzl[:n]  - D2*bzl[n:] )
#         - (D2-D1)*(D1+D2)^-1 * x[:n]
#
#     zl[:n] = D1 * ( x[:n] - x[n:] - bzl[:n] )
#     zl[n:] = D2 * (-x[:n] - x[n:] - bzl[n:] ).
#
# The first equation has the form
#
#     (A'*A + D)*x[:n]  =  rhs
#
# and is equivalent to
#
#     [ D    A' ] [ x:n] ]  = [ rhs ]
#     [ A   -I  ] [ v    ]    [ 0   ].
#
# It can be solved as
#
#     ( A*D^-1*A' + I ) * v = A * D^-1 * rhs
#     x[:n] = D^-1 * ( rhs - A'*v ).

S = matrix(0.0, (m,m))
Asc = matrix(0.0, (m,n))
v = matrix(0.0, (m,1))

def Fkkt(W):

    # Factor
    #
    #     S = A*D^-1*A' + I
    #
    # where D = 2*D1*D2*(D1+D2)^-1, D1 = d[:n]**-2, D2 = d[n:]**-2.

    d1, d2 = W['di'][:n]**2, W['di'][n:]**2

    # ds is square root of diagonal of D
    ds = math.sqrt(2.0) * div( mul( W['di'][:n], W['di'][n:]),
        base.sqrt(d1+d2) )
    d3 =  div(d2 - d1, d1 + d2)

    # Asc = A*diag(d)^-1/2
    Asc = A * spdiag(ds**-1)
```

```
        # S = I + A * D^-1 * A'
        blas.syrk(Asc, S)
        S[::m+1] += 1.0
        lapack.potrf(S)

        def g(x, y, z):

            x[:n] = 0.5 * ( x[:n] - mul(d3, x[n:]) +
                mul(d1, z[:n] + mul(d3, z[:n])) - mul(d2, z[n:] -
                mul(d3, z[n:])) )
            x[:n] = div( x[:n], ds)

            # Solve
            #
            #     S * v = 0.5 * A * D^-1 * ( bx[:n] -
            #         (D2-D1)*(D1+D2)^-1 * bx[n:] +
            #         D1 * ( I + (D2-D1)*(D1+D2)^-1 ) * bzl[:n] -
            #         D2 * ( I - (D2-D1)*(D1+D2)^-1 ) * bzl[n:] )

            blas.gemv(Asc, x, v)
            lapack.potrs(S, v)

            # x[:n] = D^-1 * ( rhs - A'*v ).
            blas.gemv(Asc, v, x, alpha=-1.0, beta=1.0, trans='T')
            x[:n] = div(x[:n], ds)

            # x[n:] = (D1+D2)^-1 * ( bx[n:] - D1*bzl[:n]  - D2*bzl[n:] )
            #         - (D2-D1)*(D1+D2)^-1 * x[:n]
            x[n:] = div( x[n:] - mul(d1, z[:n]) - mul(d2, z[n:]), d1+d2 )\
                - mul( d3, x[:n] )

            # zl[:n] = D1^1/2 * (  x[:n] - x[n:] - bzl[:n] )
            # zl[n:] = D2^1/2 * ( -x[:n] - x[n:] - bzl[n:] ).
            z[:n] = mul( W['di'][:n],  x[:n] - x[n:] - z[:n] )
            z[n:] = mul( W['di'][n:], -x[:n] - x[n:] - z[n:] )

        return g

    return solvers.coneqp(P, q, G, h, kktsolver = Fkkt)['x'][:n]
```

8.8 Optional Solvers

CVXOPT includes optional interfaces to several other optimization libraries.

GLPK `lp()` with the `solver='glpk'` option uses the simplex algorithm in GLPK (GNU Linear Programming Kit)[4].

MOSEK `lp()` and `socp()` with the `solver='mosek'` option use MOSEK[5] version 5.

[4] `http://www.gnu.org/software/glpk/glpk.html`
[5] `http://www.mosek.com`

DSDP `sdp()` with the `solver='dsdp'` option uses the DSDP5.8[6] solver.

GLPK, MOSEK and DSDP are not included in the CVXOPT distribution and need to be installed separately.

8.9 Algorithm Parameters

In this section we list some algorithm control parameters that can be modified without editing the source code. These control parameters are accessible via the dictionary `solvers.options`. By default the dictionary is empty and the default values of the parameters are used.
One can change the parameters in the default solvers by adding entries with the following key values.

`'show_progress'` `True` or `False`; turns the output to the screen on or off (default: `True`).

`'maxiters'` maximum number of iterations (default: 100).

`'abstol'` absolute accuracy (default: `1e-7`).

`'reltol'` relative accuracy (default: `1e-6`).

`'feastol'` tolerance for feasibility conditions (default: `1e-7`).

`'refinement'` number of iterative refinement steps when solving KKT equations (default: 0 if the problem has no second-order cone or matrix inequality constraints; 1 otherwise).

For example the command

```
>>> from cvxopt import solvers
>>> solvers.options['show_progress'] = False
```

turns off the screen output during calls to the solvers.
The tolerances `abstol`, `reltol` and `feastol` have the following meaning. `conelp()` terminates with status `'optimal'` if

$$s \succeq 0, \qquad z \succeq 0, \qquad \frac{\|Gx+s-h\|_2}{\max\{1,\|h\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|Ax-b\|_2}{\max\{1,\|b\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|G^Tz+A^Ty+c\|_2}{\max\{1,\|c\|_2\}} \leq \epsilon_{\text{feas}},$$

and

$$s^Tz \leq \epsilon_{\text{abs}} \qquad \text{or} \qquad \left(\min\left\{c^Tx, h^Tz+b^Ty\right\} < 0 \quad \text{and} \quad \frac{s^Tz}{-\min\{c^Tx, h^Tz+b^Ty\}} \leq \epsilon_{\text{rel}}\right).$$

It returns with status `'primal infeasible'` if

$$z \succeq 0, \qquad \frac{\|G^Tz+A^Ty\|_2}{\max\{1,\|c\|_2\}} \leq \epsilon_{\text{feas}}, \qquad h^Tz+b^Ty = -1.$$

It returns with status `'dual infeasible'` if

$$s \succeq 0, \qquad \frac{\|Gx+s\|_2}{\max\{1,\|h\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|Ax\|_2}{\max\{1,\|b\|_2\}} \leq \epsilon_{\text{feas}}, \qquad c^Tx = -1.$$

The functions `lp()`, `socp()` and `sdp()` call `conelp()` and hence use the same stopping criteria.
The function `coneqp()` terminates with status `'optimal'` if

$$s \succeq 0, \qquad z \succeq 0, \qquad \frac{\|Gx+s-h\|_2}{\max\{1,\|h\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|Ax-b\|_2}{\max\{1,\|b\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|Px+G^Tz+A^Ty+q\|_2}{\max\{1,\|q\|_2\}} \leq \epsilon_{\text{feas}},$$

[6] `http://www-unix.mcs.anl.gov/DSDP`

and

$$s^T z \leq \epsilon_{\text{abs}} \quad \text{or} \quad \left(\frac{1}{2}x^T Px + q^T x < 0, \quad \text{and} \quad \frac{s^T z}{-(1/2)x^T Px - q^T x} \leq \epsilon_{\text{rel}}\right) \quad \text{or} \quad \left(L(x,y,z) > 0 \quad \text{and} \quad \frac{s^T z}{L(x,y,z)}$$

where

$$L(x,y,z) = \frac{1}{2}x^T Px + q^T x + z^T(Gx - h) + y^T(Ax - b).$$

The function `qp()` calls `coneqp()` and hence uses the same stopping criteria.
The control parameters listed in the GLPK documentation are set to their default values and can be customized by making an entry in `solvers.options`. The keys in the dictionary are strings with the name of the GLPK parameter. For example, the command

```
>>> from cvxopt import solvers
>>> solvers.options['LPX_K_MSGLEV'] = 0
```

turns off the screen output subsequent calls `lp()` with the `'glpk'` option.
The MOSEK interior-point algorithm parameters are set to their default values. They can be modified by adding an entry `solvers.options['MOSEK']`. This entry is a dictionary with MOSEK parameter/value pairs, with the parameter names imported from `pymosek`. For details see Section 14.1.3 of the MOSEK Python API Manual[7].
For example the commands

```
>>> from cvxopt import solvers
>>> import pymosek
>>> solvers.options['MOSEK'] = {pymosek.iparam.log: 0}
```

turn off the screen output during calls of `lp()` or `socp()` with the `'mosek'` option.
The following control parameters in `solvers.options` affect the execution of the DSDP algorithm:

`'DSDP_Monitor'` the interval (in number of iterations) at which output is printed to the screen (default: 0).

`'DSDP_MaxIts'` maximum number of iterations.

`'DSDP_GapTolerance'` relative accuracy (default: `1e-5`).

[7] `http://www.mosek.com/fileadmin/products/5_0/tools/doc/html/pyapi/index.html`

Chapter 9

Nonlinear Convex Optimization (`cvxopt.solvers`)

In this chapter we consider nonlinear convex optimization problems of the form

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_k(x) \leq 0, \quad k = 1, \ldots, m \\ & Gx \preceq h \\ & Ax = b. \end{array}$$

The functions f_k convex and twice differentiabler and the linear inequalities are generalized inequalities with respect to a proper convex cone, defined as a product of a nonnegative orthant, second-order cones, and positive semidefinite cones.
The basic functions are `cp()` and `cpl()`, described in sections 9.1 and 9.2. A simpler interface for geometric programming problems is discussed in section 9.3. In section 9.4 we explain how custom solvers can be implemented that exploit structure in specific classes of problems. The last section describes the algorithm parameters that control the solvers.

9.1 General Solver

cp(`F[, G, h[, dims[, A, b[, kktsolver]]]]`)

Solves a convex optimization problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_k(x) \leq 0, \quad k = 1, \ldots, m \\ & Gx \preceq h \\ & Ax = b, \end{array} \tag{9.1}$$

The argument `F` is a function that evaluates the objective and nonlinear constraint functions. It must handle the following calling sequences.

- `F()` returns a tuple (`m`, `x0`), where `m` is the number of nonlinear constraints and `x0` is a point in the domain of f. `x0` is a dense real matrix of size (n, 1).
- `F(x)`, with `x` a dense real matrix of size (n,1), returns a tuple (`f`, `Df`). `f` is a dense real matrix of size $(m+1, 1)$, with `f[k]` equal to $f_k(x)$. (If m is zero, `f` can also be returned as a number.) `Df` is a dense or sparse real matrix of size $(m+1, n)$ with `Df[k,:]` equal to the transpose of the gradient $\nabla f_k(x)$. If `x` is not in the domain of f, `F(x)` returns `None` or a tuple (`None`,`None`).

- `F(x,z)`, with `x` a dense real matrix of size (n,1) and `z` a positive dense real matrix of size (m+1,1) returns a tuple (`f`, `Df`, `H`). `f` and `Df` are defined as above. `H` is a square dense or sparse real matrix of size (n, n), whose lower triangular part contains the lower triangular part of

$$z_0\nabla^2 f_0(x) + z_1\nabla^2 f_1(x) + \cdots + z_m\nabla^2 f_m(x).$$

  If `F` is called with two arguments, it can be assumed that x is in the domain of f.

The linear inequalities are with respect to a cone C defined as a Cartesian product of a nonnegative orthant, a number of second-order cones, and a number of positive semidefinite cones:

$$C = C_0 \times C_1 \times \cdots \times C_M \times C_{M+1} \times \cdots \times C_{M+N}$$

with

$$C_0 = \{u \in \mathbf{R}^l \mid u_k \geq 0,\ k = 1, \ldots, l\}, \qquad C_{k+1} = \{(u_0, u_1) \in \mathbf{R}\times\mathbf{R}^{r_k-1} \mid u_0 \geq \|u_1\|_2\}, \quad k = 0, \ldots, M-1, \qquad C_{k+M+1}$$

Here $\mathbf{vec}(u)$ denotes a symmetric matrix u stored as a vector in column major order.

The arguments `h` and `b` are real single-column dense matrices. `G` and `A` are real dense or sparse matrices. The default values for `A` and `b` are sparse matrices with zero rows, meaning that there are no equality constraints. The number of rows of `G` and `h` is equal to

$$K = l + \sum_{k=0}^{M-1} r_k + \sum_{k=0}^{N-1} t_k^2.$$

The columns of `G` and `h` are vectors in

$$\mathbf{R}^l \times \mathbf{R}^{r_0} \times \cdots \times \mathbf{R}^{r_{M-1}} \times \mathbf{R}^{t_0^2} \times \cdots \times \mathbf{R}^{t_{N-1}^2},$$

where the last N components represent symmetric matrices stored in column major order. The strictly upper triangular entries of these matrices are not accessed (i.e., the symmetric matrices are stored in the 'L'-type column major order used in the `blas` and `lapack` modules).

The argument `dims` is a dictionary with the dimensions of the cones. It has three fields.

`dims['l']`: l, the dimension of the nonnegative orthant (a nonnegative integer).

`dims['q']`: $[r_0, \ldots, r_{M-1}]$, a list with the dimensions of the second-order cones (positive integers).

`dims['s']`: $[t_0, \ldots, t_{N-1}]$, a list with the dimensions of the positive semidefinite cones (nonnegative integers).

The default value of `dims` is {`'l': h.size[0], 'q': [], 's': []`}, i.e., the default assumption is that the linear inequalities are componentwise inequalities.

The role of the optional argument `kktsolver` is explained in section 9.4.

`cp()` returns a dictionary with keys `'status'`, `'x'`, `'snl'`, `'sl'`, `'y'`, `'znl'`, `'zl'`. The possible values of the `'status'` key are:

`'optimal'` In this case the `'x'` entry of the dictionary is the primal optimal solution, the `'snl'` and `'sl'` entries are the corresponding slacks in the nonlinear and linear inequality constraints, and the `'znl'`, `'zl'` and `'y'` entries are the optimal values of the dual variables associated with the nonlinear inequalities, the linear inequalities, and the linear equality constraints. These vectors approximately satisfy the Karush-Kuhn-Tucker (KKT) conditions

$$\nabla f_0(x)+D\tilde{f}(x)^T z_{\mathrm{nl}}+G^T z_{\mathrm{l}}+A^T y = 0, \qquad \tilde{f}(x)+s_{\mathrm{nl}} = 0, \quad k = 1, \ldots, m, \qquad Gx+s_{\mathrm{l}} = h, \qquad Ax = b,$$

where $\tilde{f} = (f_1, \ldots, f_m)$,

$$s_{\mathrm{nl}} \succeq 0, \qquad s_{\mathrm{l}} \succeq 0, \qquad z_{\mathrm{nl}} \succeq 0, \qquad z_{\mathrm{l}} \succeq 0, \qquad s_{\mathrm{nl}}^T z_{\mathrm{nl}} + s_{\mathrm{l}}^T z_{\mathrm{l}} = 0.$$

`'unknown'` This indicates that the algorithm reached the maximum number of iterations before a solution was found. The `'x'`, `'snl'`, `'sl'`, `'y'`, `'znl'` and `'zl'` entries are `None`.

`cp()` requires that the problem is solvable and that

$$\operatorname{rank}(A) = p, \qquad \operatorname{rank}\left(\begin{bmatrix} \sum_{k=0}^m z_k \nabla^2 f_k(x) & A^T & \nabla f_1(x) & \cdots \nabla f_m(x) & G^T \end{bmatrix}\right) = n,$$

for all x and all positive z.

Example: equality constrained analytic centering The equality constrained analytic centering problem is defined as

$$\begin{array}{ll} \text{minimize} & -\sum_{i=1}^m \log x_i \\ \text{subject to} & Ax = b. \end{array}$$

The function `acent()` defined below solves the problem, assumping it is solvable.

```
from cvxopt import solvers
from cvxopt.base import matrix, spdiag, log

def acent(A, b):
    m, n = A.size
    def F(x=None, z=None):
        if x is None: return 0, matrix(1.0, (n,1))
        if min(x) <= 0.0: return None
        f = -sum(log(x))
        Df = -(x**-1).T
        if z is None: return f, Df
        H = spdiag(z[0] * x**-2)
        return f, Df, H
    return solvers.cp(F, A=A, b=b)['x']
```

Example: robust least-squares The function `robls()` defined below solves the unconstrained problem

$$\text{minimize} \quad \sum_{k=1}^m \phi((Ax-b)_k), \qquad \text{where} \quad A \in \mathbf{R}^{m\times n}, \quad \phi(u) = \sqrt{\rho + u^2}.$$

```
from cvxopt import solvers
from cvxopt.base import matrix, spdiag, sqrt, div

def robls(A, b, rho):
    m, n = A.size
    def F(x=None, z=None):
        if x is None: return 0, matrix(0.0, (n,1))
        y = A*x-b
        w = sqrt(rho + y**2)
        f = sum(w)
        Df = div(y, w).T * A
        if z is None: return f, Df
        H = A.T * spdiag(z[0]*rho*(w**-3)) * A
        return f, Df, H
    return solvers.cp(F)['x']
```

Example: analytic centering with cone constraints

$$\begin{array}{ll} \text{minimize} & -\log(1-x_1^2) - \log(1-x_2^2) - \log(1-x_3^2) \\ \text{subject to} & \|x\|_2 \leq 1 \\ & x_1 \begin{bmatrix} -21 & -11 & 0 \\ -11 & 10 & 8 \\ 0 & 8 & 5 \end{bmatrix} + x_2 \begin{bmatrix} 0 & 10 & 16 \\ 10 & -10 & -10 \\ 16 & -10 & 3 \end{bmatrix} + x_3 \begin{bmatrix} -5 & 2 & -17 \\ 2 & -6 & 8 \\ -17 & -7 & 6 \end{bmatrix} \preceq \begin{bmatrix} 20 & 10 & 40 \\ 10 & 80 & 10 \\ 40 & 10 & 15 \end{bmatrix}. \end{array}$$

```
from cvxopt.base import matrix, log, div, spdiag
from cvxopt import solvers

def F(x = None, z = None):
     if x is None:  return 0, matrix(0.0, (3,1))
     if max(abs(x)) >= 1.0:  return None
     u = 1 - x**2
     val = -sum(log(u))
     Df = div(2*x, u).T
     if z is None:  return val, Df
     H = spdiag(2 * z[0] * div(1 + u**2, u**2))
     return val, Df, H

G = matrix([ [0., -1.,  0.,  0., -21., -11.,   0., -11.,  10.,   8.,   0.,   8., 5.],
             [0.,  0., -1.,  0.,   0.,  10.,  16.,  10., -10., -10.,  16., -10., 3.],
             [0.,  0.,  0., -1.,  -5.,   2., -17.,   2.,  -6.,   8., -17.,  -7., 6.] ]
h = matrix([1.0, 0.0, 0.0, 0.0, 20., 10., 40., 10., 80., 10., 40., 10., 15.])
dims = {'l': 0, 'q': [4], 's':  [3]}
sol = solvers.cp(F, G, h, dims)
print sol['x']
[ 4.11e-01]
[ 5.59e-01]
[-7.20e-01]
```

9.2 Problems with Linear Objectives

The following function for problems with a linear objective is simpler to use than `cp()`. (In fact, `cp()` converts the problem to one with a linear objective, and then calls `cpl()`.)

cpl(`c, F[, G, h[, dims[, A, b[, kktsolver]]]]`)

Solves a convex optimization problem with a linear objective

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & f_k(x) \leq 0, \quad k = 0, \ldots, m-1 \\ & Gx \preceq h \\ & Ax = b. \end{array}$$

`c` is a real single-column dense matrix.

`F` is a function that evaluates the nonlinear constraint functions. It must handle the following calling sequences.

- `F()` returns a tuple (`m`, `x0`), where m is the number of nonlinear constraints and `x0` is a point in the domain of f. `x0` is a dense real matrix of size (n, 1).

- `F(x)`, with `x` a dense real matrix of size $(n, 1)$, returns a tuple (`f`, `Df`). `f` is a dense real matrix of size $(m, 1)$, with `f[k]` equal to $f_k(x)$. `Df` is a dense or sparse real matrix of size (m, n) with `Df[k,:]` equal to the transpose of the gradient $\nabla f_k(x)$. If `x` is not in the domain of f, `F(x)` returns `None` or a tuple (`None`,`None`).
- `F(x,z)`, with `x` a dense real matrix of size (n,1) and `z` a positive dense real matrix of size (m,1) returns a tuple (`f`, `Df`, `H`). `f` and `Df` are defined as above. `H` is a square dense or sparse real matrix of size (n,n), whose lower triangular part contains the lower triangular part of

$$z_0\nabla^2 f_0(x) + z_1\nabla^2 f_1(x) + \cdots + z_{m-1}\nabla^2 f_{m-1}(x).$$

  If `F` is called with two arguments, it can be assumed that x is in the domain of f.

The other arguments have the same meaning as in `cp()`.

`cpl()` requires that the problem is solvable and that

$$\text{rank}(A) = p, \qquad \text{rank}\left(\left[\begin{array}{ccccc} \sum_{k=0}^{m-1} z_k\nabla^2 f_k(x) & A^T & \nabla f_0(x) & \cdots \nabla f_{m-1}(x) & G^T \end{array}\right]\right) = n,$$

for all x and all positive z.

Example: floor planning This example is the floor planning problem of section 8.8.2 in the book Convex Optimization[1]:

$$\begin{array}{ll} \text{minimize} & W + H \\ \text{subject to} & A_{\min,k}/h_k - w_k \leq 0, \quad k = 1, \ldots, 5 \\ & x_1 \geq 0, \quad x_2 \geq 0, \quad x_4 \geq 0 \\ & x_1 + w_1 + \rho \leq x_3, \quad x_2 + w_2 + \rho \leq x_3, \quad x_3 + w_3 + \rho \leq x_5, \quad x_4 + w_4 + \rho \leq x_5, \quad x_5 + w_5 \leq W \\ & y_2 \geq 0, \quad y_3 \geq 0, \quad y_5 \geq 0 \\ & y_2 + h_2 + \rho \leq y_1, \quad y_1 + h_1 + \rho \leq y_4, \quad y_3 + h_3 + \rho \leq y_4, \quad y_4 + h_4 \leq H, \quad y_5 + h_5 \leq H \\ & h_k/\gamma \leq w_k \leq \gamma h_k, \quad k = 1, \ldots, 5. \end{array}$$

This problem has 22 variables

$$W, \qquad H, \qquad x \in \mathbf{R}^5, \qquad y \in \mathbf{R}^5, \qquad w \in \mathbf{R}^5, \qquad h \in \mathbf{R}^5,$$

5 nonlinear inequality constraints, and 26 linear inequality constraints. The code belows defines a function `floorplan()` that solves the problem by calling `cp()`, then applies it to 4 instances, and creates a figure.

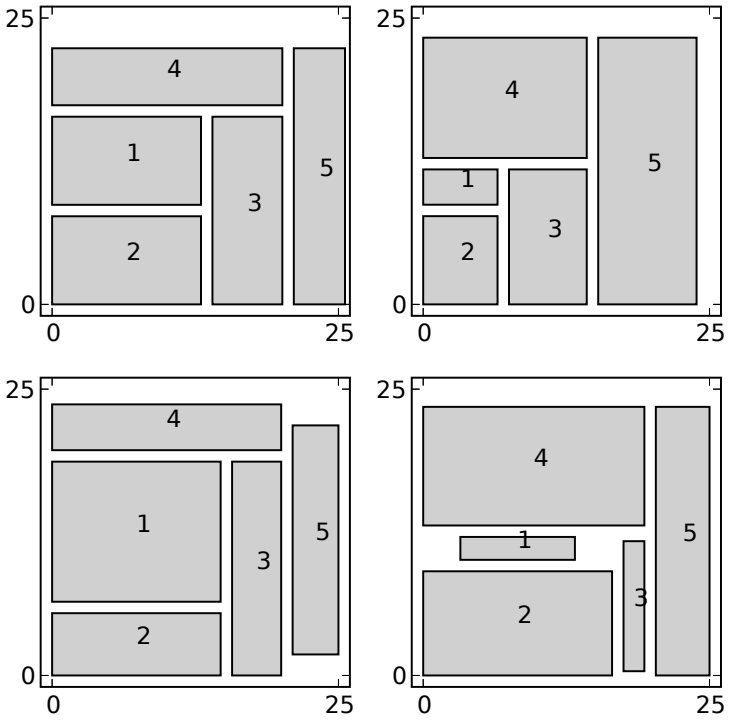
```
import pylab
from cvxopt import solvers
from cvxopt.base import matrix, spmatrix, mul, div

def floorplan(Amin):

    #     minimize    W+H
    #     subject to  Amink / hk <= wk, k = 1,..., 5
    #                 x1 >= 0,  x2 >= 0, x4 >= 0
    #                 x1 + w1 + rho <= x3
    #                 x2 + w2 + rho <= x3
    #                 x3 + w3 + rho <= x5
    #                 x4 + w4 + rho <= x5
    #                 x5 + w5 <= W
    #                 y2 >= 0,  y3 >= 0,  y5 >= 0
```

[1] `http://www.stanford.edu/~{}boyd/cvxbook`

```
#                 y2 + h2 + rho <= y1
#                 y1 + h1 + rho <= y4
#                 y3 + h3 + rho <= y4
#                 y4 + h4 <= H
#                 y5 + h5 <= H
#                 hk/gamma <= wk <= gamma*hk,  k = 1, ..., 5
#
# 22 Variables W, H, x (5), y (5), w (5), h (5).
#
# W, H:  scalars; bounding box width and height
# x, y:  5-vectors; coordinates of bottom left corners of blocks
# w, h:  5-vectors; widths and heigths of the 5 blocks

rho, gamma = 1.0, 5.0   # min spacing, min aspect ratio

# The objective is to minimize W + H.  There are five nonlinear
# constraints
#
#     -wk + Amink / hk <= 0,  k = 1, ..., 5

c = matrix(2*[1.0] + 20*[0.0])

def F(x=None, z=None):
    if x is None:  return 5, matrix(17*[0.0] + 5*[1.0])
    if min(x[17:]) <= 0.0:  return None
    f = -x[12:17] + div(Amin, x[17:])
    Df = matrix(0.0, (5,22))
    Df[:,12:17] = spmatrix(-1.0, range(5), range(5))
    Df[:,17:] = spmatrix(-div(Amin, x[17:]**2), range(5), range(5))
    if z is None: return f, Df
    H = spmatrix( 2.0* mul(z, div(Amin, x[17::]**3)), range(17,22), range(17,22) )
    return f, Df, H

G = matrix(0.0, (26,22))
h = matrix(0.0, (26,1))
G[0,2] = -1.0                                          # -x1 <= 0
G[1,3] = -1.0                                          # -x2 <= 0
G[2,5] = -1.0                                          # -x4 <= 0
G[3, [2, 4, 12]], h[3] = [1.0, -1.0, 1.0], -rho        # x1 - x3 + w1 <= -rho
G[4, [3, 4, 13]], h[4] = [1.0, -1.0, 1.0], -rho        # x2 - x3 + w2 <= -rho
G[5, [4, 6, 14]], h[5] = [1.0, -1.0, 1.0], -rho        # x3 - x5 + w3 <= -rho
G[6, [5, 6, 15]], h[6] = [1.0, -1.0, 1.0], -rho        # x4 - x5 + w4 <= -rho
G[7, [0, 6, 16]] = -1.0, 1.0, 1.0                      # -W + x5 + w5 <= 0
G[8,8] = -1.0                                          # -y2 <= 0
G[9,9] = -1.0                                          # -y3 <= 0
G[10,11] = -1.0                                        # -y5 <= 0
G[11, [7, 8, 18]], h[11] = [-1.0, 1.0, 1.0], -rho      # -y1 + y2 + h2 <= -rho
G[12, [7, 10, 17]], h[12] = [1.0, -1.0, 1.0], -rho     #  y1 - y4 + h1 <= -rho
G[13, [9, 10, 19]], h[13] = [1.0, -1.0, 1.0], -rho     #  y3 - y4 + h3 <= -rho
G[14, [1, 10, 20]] = -1.0, 1.0, 1.0                    # -H + y4 + h4 <= 0
G[15, [1, 11, 21]] = -1.0, 1.0, 1.0                    # -H + y5 + h5 <= 0
G[16, [12, 17]] = -1.0, 1.0/gamma                      # -w1 + h1/gamma <= 0
```

```
    G[17, [12, 17]] = 1.0, -gamma                          #  w1 - gamma * h1 <= 0
    G[18, [13, 18]] = -1.0, 1.0/gamma                      # -w2 + h2/gamma <= 0
    G[19, [13, 18]] = 1.0, -gamma                          #  w2 - gamma * h2 <= 0
    G[20, [14, 18]] = -1.0, 1.0/gamma                      # -w3 + h3/gamma <= 0
    G[21, [14, 19]] = 1.0, -gamma                          #  w3 - gamma * h3 <= 0
    G[22, [15, 19]] = -1.0, 1.0/gamma                      # -w4  + h4/gamma <= 0
    G[23, [15, 20]] = 1.0, -gamma                          #  w4 - gamma * h4 <= 0
    G[24, [16, 21]] = -1.0, 1.0/gamma                      # -w5 + h5/gamma <= 0
    G[25, [16, 21]] = 1.0, -gamma                          #  w5 - gamma * h5 <= 0.0

    # solve and return W, H, x, y, w, h
    sol = solvers.cpl(c, F, G, h)
    return  sol['x'][0], sol['x'][1], sol['x'][2:7], sol['x'][7:12], sol['x'][12:17],

pylab.figure(facecolor='w')
pylab.subplot(221)
Amin = matrix([100., 100., 100., 100., 100.])
W, H, x, y, w, h =  floorplan(Amin)
for k in xrange(5):
    pylab.fill([x[k], x[k], x[k]+w[k], x[k]+w[k]],
               [y[k], y[k]+h[k], y[k]+h[k], y[k]], facecolor = '#D0D0D0')
    pylab.text(x[k]+.5*w[k], y[k]+.5*h[k], "%d" %(k+1))
pylab.axis([-1.0, 26, -1.0, 26])
pylab.xticks([])
pylab.yticks([])

pylab.subplot(222)
Amin = matrix([20., 50., 80., 150., 200.])
W, H, x, y, w, h =  floorplan(Amin)
for k in xrange(5):
    pylab.fill([x[k], x[k], x[k]+w[k], x[k]+w[k]],
               [y[k], y[k]+h[k], y[k]+h[k], y[k]], 'facecolor = #D0D0D0')
    pylab.text(x[k]+.5*w[k], y[k]+.5*h[k], "%d" %(k+1))
pylab.axis([-1.0, 26, -1.0, 26])
pylab.xticks([])
pylab.yticks([])

pylab.subplot(223)
Amin = matrix([180., 80., 80., 80., 80.])
W, H, x, y, w, h =  floorplan(Amin)
for k in xrange(5):
    pylab.fill([x[k], x[k], x[k]+w[k], x[k]+w[k]],
               [y[k], y[k]+h[k], y[k]+h[k], y[k]], 'facecolor = #D0D0D0')
    pylab.text(x[k]+.5*w[k], y[k]+.5*h[k], "%d" %(k+1))
pylab.axis([-1.0, 26, -1.0, 26])
pylab.xticks([])
pylab.yticks([])

pylab.subplot(224)
Amin = matrix([20., 150., 20., 200., 110.])
W, H, x, y, w, h =  floorplan(Amin)
for k in xrange(5):
```

```
    pylab.fill([x[k], x[k], x[k]+w[k], x[k]+w[k]],
               [y[k], y[k]+h[k], y[k]+h[k], y[k]], 'facecolor = #D0D0D0')
    pylab.text(x[k]+.5*w[k], y[k]+.5*h[k], "%d" %(k+1))
pylab.axis([-1.0, 26, -1.0, 26])
pylab.xticks([])
pylab.yticks([])

pylab.show()
```



9.3 Geometric Programming

gp(`K, F, g [, G, h [, A, b]]`)

Solves a geometric program in convex form

$$\begin{array}{ll} \text{minimize} & f_0(x) = \mathbf{lse}(F_0 x + g_0) \\ \text{subject to} & f_i(x) = \mathbf{lse}(F_i x + g_i) \leq 0, \quad i = 1, \ldots, m \\ & Gx \preceq h \\ & Ax = b \end{array}$$

where

$$\mathbf{lse}(u) = \log \sum_k \exp(u_k), \qquad F = \begin{bmatrix} F_0^T & F_1^T & \cdots & F_m^T \end{bmatrix}^T, \qquad g = \begin{bmatrix} g_0^T & g_1^T & \cdots & g_m^T \end{bmatrix}^T.$$

`K` is a list of $m+1$ positive integers with `K[i]` equal to the number of rows in F_i. `F` is a dense or sparse real matrix of size `(sum(K),n)`. `g` is a dense real matrix with one column and the same number of rows as `F`. `G` and `A` are dense or sparse real matrices. Their default values are sparse matrices with zero rows. `h` and `b` are dense real matrices with one column. Their default values are matrices of size (0,1).

`gp()` returns a dictionary with keys `'status'`, `'x'`, `'snl'`, `'sl'`, `'y'`, `'znl'` and `'zl'`. The possible values of the `'status'` key are:

`'optimal'` In this case the `'x'` entry is the primal optimal solution, the `'snl'` and `'sl'` entries are the corresponding slacks in the nonlinear and linear inequality constraints. The `'znl'`, `'zl'` and `'y'` entries are the optimal values of the dual variables associated with the nonlinear and linear inequality constraints and the linear equality constraints. These values approximately satisfy

$$\nabla f_0(x)+\sum_{k=1}^{m} z_{\mathrm{nl},k}\nabla f_k(x)+G^T z_{\mathrm{l}}+A^T y = 0, \qquad f_k(x)+s_{\mathrm{nl},k} = 0, \quad k=1,\ldots,m, \qquad Gx+s_{\mathrm{l}} = h, \qquad Ax = b$$

and

$$s_{\mathrm{nl}} \succeq 0, \qquad s_{\mathrm{l}} \succeq 0, \qquad z_{\mathrm{nl}} \succeq 0, \qquad z_{\mathrm{l}} \succeq 0, \qquad s_{\mathrm{nl}}^T z_{\mathrm{nl}} + s_{\mathrm{l}}^T z_{\mathrm{l}} = 0.$$

`'unknown'` This means that the algorithm reached the maximum number of iterations before a solution was found. The `'x'`, `'snl'`, `'sl'`, `'y'`, `'znl'` and `'zl'` entries are `None`.

As an example, we solve the small GP of section 2.4 of the paper A Tutorial on Geometric Programming[2]. The posynomial form of the problem is

$$\begin{array}{ll} \text{minimize} & w^{-1}h^{-1}d^{-1} \\ \text{subject to} & (2/A_{\mathrm{wall}})hw + (2/A_{\mathrm{wall}})hd \leq 1 \\ & (1/A_{\mathrm{flr}})wd \leq 1 \\ & \alpha wh^{-1} \leq 1 \\ & (1/\beta)hw^{-1} \leq 1 \\ & \gamma wd^{-1} \leq 1 \\ & (1/\delta)dw^{-1} \leq 1 \end{array}$$

with variables h, w, d.

```
from cvxopt.base import matrix, log, exp
from cvxopt import solvers

Aflr  = 1000.0
Awall = 100.0
alpha = 0.5
beta  = 2.0
gamma = 0.5
delta = 2.0

F = matrix( [[-1., 1., 1., 0., -1.,  1.,  0.,  0.],
             [-1., 1., 0., 1.,  1., -1.,  1., -1.],
             [-1., 0., 1., 1.,  0.,  0., -1.,  1.]])
g = log( matrix( [1.0, 2/Awall, 2/Awall, 1/Aflr, alpha, 1/beta, gamma, 1/delta]) )
K = [1, 2, 1, 1, 1, 1, 1]
h, w, d = exp( solvers.gp(K, F, g)['x'] )
```

[2] http://www.stanford.edu/~{}boyd/gp_tutorial.html

9.4 Exploiting Structure

By default, the functions `cp()` and `cpl()` do not exploit problem structure. Two mechanisms are provided for implementing customized solvers that take advantage of problem structure.

Providing a function for solving KKT equations. The most expensive step of each iteration of `cp()` is the solution of a set of linear equations ('KKT equations') of the form

$$\begin{bmatrix} H & A^T & \tilde{G}^T \\ A & 0 & 0 \\ \tilde{G} & 0 & -W^TW \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}, \tag{9.2}$$

where

$$H = \sum_{k=0}^{m} z_k \nabla^2 f_k(x), \qquad \tilde{G} = \begin{bmatrix} \nabla f_1(x) & \cdots & \nabla f_m(x) & G^T \end{bmatrix}^T.$$

The matrix W depends on the current iterates and is defined as follows. Suppose

$$u = (u_{\mathrm{nl}},\ u_{\mathrm{l}},\ u_{\mathrm{q},0},\ \ldots,\ u_{\mathrm{q},M-1},\ \mathbf{vec}\,(u_{\mathrm{s},0}),\ \ldots,\ \mathbf{vec}\,(u_{\mathrm{s},N-1})), \qquad u_{\mathrm{l}} \in \mathbf{R}^l, \qquad u_{\mathrm{q},k} \in \mathbf{R}^{r_k}, \quad k = 0, \ldots, M-1, \qquad u_{\mathrm{s},}$$

Then W is a block-diagonal matrix,

$$Wu = (W_{\mathrm{nl}}u_{\mathrm{nl}},\ W_{\mathrm{l}}u_{\mathrm{l}},\ W_{\mathrm{q},0}u_{\mathrm{q},0},\ \ldots,\ W_{\mathrm{q},M-1}u_{\mathrm{q},M-1},\ W_{\mathrm{s},0}\,\mathbf{vec}\,(u_{\mathrm{s},0}),\ \ldots,\ W_{\mathrm{s},N-1}\,\mathbf{vec}\,(u_{\mathrm{s},N-1}))$$

with the following diagonal blocks.

- The first block is a *positive diagonal scaling* with a vector d_{nl}:

  $$W_{\mathrm{nl}} = \mathbf{diag}\,(d_{\mathrm{nl}}), \qquad W_{\mathrm{nl}}^{-1} = \mathbf{diag}\,(d_{\mathrm{nl}})^{-1}.$$

  This transformation is symmetric:

  $$W_{\mathrm{nl}}^T = W_{\mathrm{nl}}.$$

- The second block is a *positive diagonal scaling* with a vector d_{l}:

  $$W_{\mathrm{l}} = \mathbf{diag}\,(d_{\mathrm{l}}), \qquad W_{\mathrm{l}}^{-1} = \mathbf{diag}\,(d_{\mathrm{l}})^{-1}.$$

  This transformation is symmetric:

  $$W_{\mathrm{l}}^T = W_{\mathrm{l}}.$$

- The next M blocks are positive multiples of *hyperbolic Householder transformations*:

  $$W_{\mathrm{q},k} = \beta_k(2v_kv_k^T - J), \qquad W_{\mathrm{q},k}^{-1} = \frac{1}{\beta_k}(2Jv_kv_k^TJ - J), \qquad k = 0, \ldots, M-1,$$

  where

  $$\beta_k > 0, \qquad v_{k0} > 0, \qquad v_k^TJv_k = 1, \qquad J = \begin{bmatrix} 1 & 0 \\ 0 & -I \end{bmatrix}.$$

  These transformations are also symmetric:

  $$W_{\mathrm{q},k}^T = W_{\mathrm{q},k}.$$

- The last N blocks are *congruence transformations* with nonsingular matrices:

  $$W_{\mathrm{s},k}\,\mathbf{vec}\,(u_{\mathrm{s},k}) = \mathbf{vec}\,(r_k^Tu_{\mathrm{s},k}r_k), \qquad W_{\mathrm{s},k}^{-1}\,\mathbf{vec}\,(u_{\mathrm{s},k}) = \mathbf{vec}\,(r_k^{-T}u_{\mathrm{s},k}r_k^{-1}), \qquad k = 0, \ldots, N-1.$$

  In general, this operation is not symmetric, and

  $$W_{\mathrm{s},k}^T\,\mathbf{vec}\,(u_{\mathrm{s},k}) = \mathbf{vec}\,(r_ku_{\mathrm{s},k}r_k^T), \qquad W_{\mathrm{s},k}^{-T}\,\mathbf{vec}\,(u_{\mathrm{s},k}) = \mathbf{vec}\,(r_k^{-1}u_{\mathrm{s},k}r_k^{-T}), \qquad k = 0, \ldots, N-1.$$

It is often possible to exploit problem structure to solve (9.2) faster than by standard methods. The last argument `kktsolver` of `cp()` allows the user to supply a Python function for solving the KKT equations. This function will be called as "`f = kktsolver(x, z, W)`". The argument `x` is the point at which the derivatives in the KKT matrix are evaluated. `z` is a positive vector of length it $m + 1$, containing the coefficients in the 1,1 block H. `W` is a dictionary that contains the parameters of the scaling:

- `W['dnl']` is the positive vector that defines the diagonal scaling for the nonlinear inequalities. `W['dnli']` is its componentwise inverse.
- `W['d']` is the positive vector that defines the diagonal scaling for the componentwise linear inequalities. `W['di']` is its componentwise inverse.
- `W['beta']` and `W['v']` are lists of length M with the coefficients and vectors that define the hyperbolic Householder transformations.
- `W['r']` is a list of length N with the matrices that define the the congruence transformations. `W['rti']` is a list of length N with the transposes of the inverses of the matrices in `W['r']`.

The function call "`f = kktsolver(x, z, W)`" should return a routine for solving the KKT system (9.2) defined by `x`, `z`, `W`. It will be called as "`f(bx, by, bz)`". On entry, `bx`, `by`, `bz` contain the righthand side. On exit, they should contain the solution of the KKT system, with the last component scaled, i.e., on exit,

$$b_x := u_x, \qquad b_y := u_y, \qquad b_z := Wu_z.$$

The role of the argument `kktsolver` in the function `cpl()` is similar, except that in (9.2),

$$H = \sum_{k=0}^{m-1} z_k \nabla^2 f_k(x), \qquad \tilde{G} = \begin{bmatrix} \nabla f_0(x) & \cdots & \nabla f_{m-1}(x) & G^T \end{bmatrix}^T.$$

Specifying constraints via Python functions. In the default use of `cp()`, the arguments `G` and `A` are the coefficient matrices in the constraints of (9.2). It is also possible to specify these matrices by providing Python functions that evaluate the corresponding matrix-vector products and their adjoints.

If the argument `G` of `conelp()` is a Python function, it should be defined as follows:

`G(u, v [, alpha[, beta[, trans]]])`

This evaluates the matrix-vector products

$$v := \alpha G u + \beta v \quad (\text{trans} = \text{'N'}), \qquad v := \alpha G^T u + \beta v \quad (\text{trans} = \text{'T'}).$$

The default values of the optional arguments must be `alpha = 1.0`, `beta = 0.0`, `trans = 'N'`.

Similarly, if the argument `A` is a Python function, then it must be defined as follows.

`A(u, v [, alpha[, beta[, trans]]])`

This evaluates the matrix-vector products

$$v := \alpha A u + \beta v \quad (\text{trans} = \text{'N'}), \qquad v := \alpha A^T u + \beta v \quad (\text{trans} = \text{'T'}).$$

The default values of the optional arguments must be `alpha = 1.0`, `beta = 0.0`, `trans = 'N'`.

In a similar way, when the first argument `F()` of `cp()` returns matrices of first derivatives or second derivatives `Df`, `H`, these matrices can be specified as Python functions. If `Df` is a Python function, it should be defined as follows:

`Df(u, v [, alpha[, beta[, trans]]])`

This evaluates the matrix-vector products

$$v := \alpha Df(x)u + \beta v \quad (\text{trans} = \text{'N'}), \qquad v := \alpha Df(x)^T u + \beta v \quad (\text{trans} = \text{'T'}).$$

The default values of the optional arguments must be `alpha = 1.0`, `beta = 0.0`, `trans = 'N'`.

If `H` is a Python function, it should be defined as follows:

`H(u, v [, alpha[, beta]])`

This evaluates the matrix-vector product

$$v := \alpha Hu + \beta v.$$

The default values of the optional arguments must be `alpha = 1.0`, `beta = 0.0`, `trans = 'N'`.

If `G`, `A`, `Df`, or `H` are Python functions, then the argument `kktsolver` must also be provided.

As an example, we consider the unconstrained problem

$$\text{minimize} \quad (1/2)\|Ax - b\|_2^2 - \sum_{i=1}^n \log(1 - x_i^2)$$

where A is an m by n matrix with m less than n. The Hessian of the objective is diagonal plus a low-rank term:

$$H = A^T A + \mathbf{diag}\,(d), \qquad d_i = \frac{2(1 + x_i^2)}{(1 - x_i^2)^2}.$$

We can exploit this property when solving (9.2) by applying the matrix inversion lemma. We first solve

$$(A\mathbf{diag}\,(d)^{-1}A^T + I)v = (1/z_0)A\mathbf{diag}\,(d)^{-1}b_x,$$

and then obtain

$$u_x = \mathbf{diag}\,(d)^{-1}(b_x/z_0 - A^T v).$$

The following code follows this method. It also uses BLAS functions for matrix-matrix and matrix-vector products.

```
from cvxopt.base import matrix, spdiag, mul, div, log
from cvxopt import base, blas, lapack, solvers, base

def l2ac(A, b):
    """
    Solves

        minimize  (1/2) * ||A*x-b||_2^2 - sum log (1-xi^2)

    assuming A is m x n with m << n.
    """

    m, n = A.size
    def F(x = None, z = None):
        if x is None:
            return 0, matrix(0.0, (n,1))
        if max(abs(x)) >= 1.0:
            return None
        # r = A*x - b
        r = -b
        blas.gemv(A, x, r, beta = -1.0)
```

```
        w = x**2
        f = 0.5 * blas.nrm2(r)**2  - sum(log(1-w))
        # gradf = A'*r + 2.0 * x ./ (1-w)
        gradf = div(x, 1.0 - w)
        blas.gemv(A, r, gradf, trans = 'T', beta = 2.0)
        if z is None:
            return f, gradf.T
        else:
            def Hf(u, v, alpha = 1.0, beta = 0.0):
               # v := alpha * (A'*A*u + 2*((1+w)./(1-w)).*u + beta *v
               v *= beta
               v += 2.0 * alpha * mul(div(1.0+w, (1.0-w)**2), u)
               blas.gemv(A, u, r)
               blas.gemv(A, r, v, alpha = alpha, beta = 1.0, trans = 'T')
            return f, gradf.T, Hf


    # Custom solver for the Newton system
    #
    #     z[0]*(A'*A + D)*x = bx
    #
    # where D = 2 * (1+x.^2) ./ (1-x.^2).^2.  We apply the matrix inversion
    # lemma and solve this as
    #
    #     (A * D^-1 *A' + I) * v = A * D^-1 * bx / z[0]
    #     D * x = bx / z[0] - A'*v.

    S = matrix(0.0, (m,m))
    Asc = matrix(0.0, (m,n))
    v = matrix(0.0, (m,1))
    def Fkkt(x, z, W):
        ds = (2.0 * div(1 + x**2, (1 - x**2)**2))**-0.5
        base.gemm(A, spdiag(ds), Asc)
        blas.syrk(Asc, S)
        S[::m+1] += 1.0
        lapack.potrf(S)
        a = z[0]
        def g(x, y, z):
            x[:] = mul(x, ds) / a
            blas.gemv(Asc, x, v)
            lapack.potrs(S, v)
            blas.gemv(Asc, v, x, alpha = -1.0, beta = 1.0, trans = 'T')
            x[:] = mul(x, ds)
        return g

    return solvers.cp(F, kktsolver = Fkkt)['x']
```

9.5 Algorithm Parameters

The following algorithm control parameters are accessible via the dictionary `solvers.options`. By default the dictionary is empty and the default values of the parameters are used.

One can change the parameters in the default solvers by adding entries with the following key values.

`'show_progress'` `True` or `False`; turns the output to the screen on or off (default: `True`).

`'maxiters'` maximum number of iterations (default: 100).

`'abstol'` absolute accuracy (default: `1e-7`).

`'reltol'` relative accuracy (default: `1e-6`).

`'feastol'` tolerance for feasibility conditions (default: `1e-7`).

`'refinement'` number of iterative refinement steps when solving KKT equations (default: 1).

For example the command

```
>>> from cvxopt import solvers
>>> solvers.options['show_progress'] = False
```

turns off the screen output during calls to the solvers. The tolerances `abstol`, `reltol` and `feastol` have the following meaning in `nlcp()`.
`cpl()` returns with status `'optimal'` if

$$\frac{\|c + Df(x)^T z_{\rm nl} + G^T z_{\rm l} + A^T y\|_2}{\max\{1, \|c + Df(x_0)^T \mathbf{1} + G^T \mathbf{1}\|_2\}} \leq \epsilon_{\rm feas}, \qquad \frac{\|(f(x) + s_{\rm nl}, Gx + s_{\rm l} - h, Ax - b)\|_2}{\max\{1, \|(f(x_0) + \mathbf{1}, Gx_0 + \mathbf{1} - h, Ax_0 - b)\|_2\}} \leq \epsilon_{\rm feas}$$

where x_0 is the point returned by `F()`, and

$$\text{gap} \leq \epsilon_{\rm abs} \qquad \text{or} \qquad \left(c^T x < 0, \quad \frac{\text{gap}}{-c^T x} \leq \epsilon_{\rm rel}\right) \qquad \text{or} \qquad \left(L(x,y,z) > 0, \quad \frac{\text{gap}}{L(x,y,z)} \leq \epsilon_{\rm rel}\right)$$

where

$$\text{gap} = \begin{bmatrix} s_{\rm nl} \\ s_{\rm l} \end{bmatrix}^T \begin{bmatrix} z_{\rm nl} \\ z_{\rm l} \end{bmatrix}, \qquad L(x,y,z) = c^T x + z_{\rm nl}^T f(x) + z_{\rm l}^T (Gx - h) + y^T (Ax - b).$$

The functions `cp()`, `qp()`, and `gp()` call `cpl()` and hence use the same stopping criteria (with $x_0 = 0$ for `qp()` and `gp()`).
The MOSEK interior-point algorithm parameters are set to their default values. They can be modified by adding an entry `solvers.options['MOSEK']`. This entry is a dictionary with MOSEK parameter/value pairs, with the parameter names imported from `pymosek`. For details see Section 14.1.3 of the MOSEK Python API Manual[3].
For example the commands

```
>>> from cvxopt import solvers
>>> import pymosek
>>> solvers.options['MOSEK'] = {pymosek.iparam.log: 0}
```

turn off the screen output during calls of `qp()` with the `'mosek'` option.

[3] `http://www.mosek.com/fileadmin/products/5_0/tools/doc/html/pyapi/index.html`

Chapter 10

Modeling (cvxopt.modeling)

The module cvxopt.modeling can be used to specify and solve optimization problems with convex piecewise-linear objective and constraint functions.
To specify an optimization problem one first defines the optimization variables (see section 10.1), and then defines the objective and constraint functions using linear operations (vector addition and subtraction, matrix-vector multiplication, indexing and slicing) and nested evaluations of max(), min(), abs() and sum() (see section 10.2).

10.1 Variables

Optimization variables are represented by variable objects.
variable([size[, name]])

A vector variable. The first argument is the dimension of the vector (a positive integer with default value 1). The second argument is a string with a name for the variable. The name is optional and has default value "". It is only used when displaying variables (or objects that depend on variables, such as functions or constraints) using print statements, when calling the built-in functions repr() or str(), or when writing linear programs to MPS files.

The function len() returns the length of a variable. A variable x has two attributes.
name

The name of the variable.

value

Either None or a dense 'd' matrix of size len(x) by 1.

The attribute x.value is set to None when the variable x is created. It can be given a numerical value later, typically by solving an LP that has x as one of its variables. One can also make an explicit assignment x.value = y. The assigned value y must be an integer or float, or a dense 'd' matrix of size (len(x),1). If y is an integer or float all the elements of x.value are set to the value of y.

```
>>> from cvxopt.base import matrix
>>> from cvxopt.modeling import variable
>>> x = variable(3,'a')
>>> len(x)
3
>>> print x.name
a
>>> print x.value
```

```
None
>>> x.value = matrix([1.,2.,3.])
>>> print x.value
[ 1.00e+00]
[ 2.00e+00]
[ 3.00e+00]
>>> x.value = 1
>>> print x.value
[ 1.00e+00]
[ 1.00e+00]
[ 1.00e+00]
```

10.2 Functions

Objective and constraint functions can be defined via overloaded operations on variables and other functions. A function `f` is interpreted as a column vector, with length `len(f)` and with a value that depends on the values of its variables. Functions have two public attributes.

variables()

Returns a copy of the list of variables of the function.

value()

The function value. If any of the variables of `f` has value `None`, then `f.value()` returns `None`. Otherwise, it returns a dense `'d'` matrix of size `(len(f),1)` with the function value computed from the `value` attributes of the variables of `f`.

Three types of functions are supported: affine, convex piecewise-linear and concave piecewise-linear.

Affine functions represent vector valued functions of the form

$$f(x_1, \ldots, x_n) = A_1 x_1 + \cdots + A_n x_n + b.$$

The coefficients can be scalars or dense or sparse matrices. The constant term is a scalar or a column vector.
Affine functions result from the following operations.

Unary operations For a variable `x`, the unary operation `+x` results in an affine function with `x` as variable, coefficient 1.0, and constant term 0.0. The unary operation `-x` returns an affine function with `x` as variable, coefficient -1.0, and constant term 0.0. For an affine function `f`, `+f` is a copy of `f`, and `-f` is a copy of `f` with the signs of its coefficients and constant term reversed.

Addition and subtraction Sums and differences of affine functions, variables and constants result in new affine functions. The constant terms in the sum can be of type `integer` or `float`, or dense or sparse `'d'` matrices with one column.

The rules for addition and subtraction follow the conventions for matrix addition and subtraction in sections 2.3 and 6.3, with variables and affine functions interpreted as dense `'d'` matrices with one column. In particular, a scalar term (`integer`, `float`, 1 by 1 dense `'d'` matrix, variable of length 1, or affine function of length 1) can be added to an affine function or variable of length greater than 1.

Multiplication Suppose `v` is an affine function or a variable, and `a` is an `integer`, `float`, sparse or dense `'d'` matrix. The products `a*v` and `v*a` are valid affine functions whenever the product is allowed under the rules for matrix and scalar multiplication of sections 2.3 and 6.3, with `v` interpreted as a `'d'` matrix with one column. In particular, the product `a*v` is defined if `a` is a scalar (`integer`, `float` or 1 by 1 dense `'d'` matrix), or a matrix (dense or sparse) with `a.size[1] = len(v)`. The operation `v*a` is defined if `a` is scalar, or if `len(v)` = 1 and `a` is a matrix with one column.

Inner products The following two functions return scalar affine functions defined as inner products of a constant vector with a variable or affine function.

sum(v)

The argument is an affine function or a variable. The result is an affine function of length 1, with the sum of the components of the argument v.

dot(u,v)

If v is a variable or affine function and u is a 'd' matrix of size (len(v),1), then dot(u,v) and dot(v,u) are equivalent to u.trans()*v.

If u and v are dense matrices, then dot(u,v) is equivalent to the function blas.dot(u,v) defined in section 3.2, i.e., it returns the inner product of the two matrices.

In the following example, the variable x has length 1 and y has length 2. The functions f and g are given by

$$\begin{aligned} f(x,y) &= \begin{bmatrix} 2 \\ 2 \end{bmatrix} x + y + \begin{bmatrix} 3 \\ 3 \end{bmatrix}, \\ g(x,y) &= \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} f(x,y) + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} y + \begin{bmatrix} 1 \\ -1 \end{bmatrix} \\ &= \begin{bmatrix} 8 \\ 12 \end{bmatrix} x + \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} y + \begin{bmatrix} 13 \\ 17 \end{bmatrix}. \end{aligned}$$

```
>>> from cvxopt.modeling import variable
>>> x = variable(1,'x')
>>> y = variable(2,'y')
>>> f = 2*x + y + 3
>>> A = matrix([[1., 2.], [3.,4.]])
>>> b = matrix([1.,-1.])
>>> g = A*f + sum(y) + b
>>> print g
affine function of length 2
constant term:
[ 1.30e+01]
[ 1.70e+01]
linear term: linear function of length 2
coefficient of variable(2,'y'):
[ 2.00e+00  4.00e+00]
[ 3.00e+00  5.00e+00]
coefficient of variable(1,'x'):
[ 8.00e+00]
[ 1.20e+01]
```

In-place operations For an affine function f the operations f += u and f -= u, with u a constant, a variable or an affine function, are allowed if they do not change the length of f, i.e., if u has length len(f) or length 1. In-place multiplication f *= u and division f /= u are allowed if u is an integer, float, or 1 by 1 matrix.

Indexing and slicing Variables and affine functions admit single-argument indexing of the four types described in section 2.4. The result of an indexing or slicing operation is an affine function.

```
>>> x = variable(4,'x')
>>> f = x[::2]
```

```
>>> print f
linear function of length 2
linear term: linear function of length 2
coefficient of variable(4,'x'):
[ 1.00e+00     0         0         0    ]
[    0         0      1.00e+00     0    ]
>>> y = variable(3,'x')
>>> g = matrix(range(12),(3,4),'d')*x - 3*y + 1
>>> print g[0] + g[2]
affine function of length 1
constant term:
[ 2.00e+00]
linear term: linear function of length 1
coefficient of variable(4,'x'):
[ 2.00e+00  8.00e+00  1.40e+01  2.00e+01]
coefficient of variable(3,'x'):
[-3.00e+00     0     -3.00e+00]
```

The general expression of a **convex piecewise-linear** function is

$$f(x_1, \ldots, x_n) = b + A_1 x_1 + \cdots + A_n x_n + \sum_{k=1}^{K} \max(y_1, y_2, \ldots, y_{m_k}).$$

The maximum in this expression is a componentwise maximum of its vector arguments, which can be constant vectors, variables, affine functions or convex piecewise-linear functions. The general expression for a **concave piecewise-linear** function is

$$f(x_1, \ldots, x_n) = b + A_1 x_1 + \cdots + A_n x_n + \sum_{k=1}^{K} \min(y_1, y_2, \ldots, y_{m_k}).$$

Here the arguments of the `min()` can be constants, variables, affine functions or concave piecewise-linear functions. Piecewise-linear functions can be created using the following operations.

Maximum If the arguments in `f = max(y1,y2, ...)` do not include any variables or functions, then the Python built-in `max()` is evaluated.

If one or more of the arguments are variables or functions, `max()` returns a piecewise-linear function defined as the elementwise maximum of its arguments. In other words, `f[k] = max(y1[k],y2[k], ...)` for `k`=0, ..., `len(f)-1`. The length of `f` is equal to the maximum of the lengths of the arguments. Each argument must have length equal to `len(f)` or length one. Arguments with length one are interpreted as vectors of length `len(f)` with identical entries.

The arguments can be scalars of type `integer` or `float`, dense `'d'` matrices with one column, variables, affine functions or convex piecewise-linear functions.

With one argument, `f = max(u)` is interpreted as `f = max(u[0],u[1],..., u[len(u)-1])`.

Minimum Similar to `max()` but returns a concave piecewise-linear function. The arguments can be scalars of type `integer` or `float`, dense `'d'` matrices with one column, variables, affine functions or concave piecewise-linear functions.

Absolute value If `u` is a variable or affine function then `f = abs(u)` returns the convex piecewise-linear function `max(u,-u)`.

Unary plus and minus `+f` creates a copy of `f`. `-f` is a concave piecewise-linear function if `f` is convex and a convex piecewise-linear function if `f` is concave.

Addition and subtraction Sums and differences involving piecewise-linear functions are allowed if they result in convex or concave functions. For example, one can add two convex or two concave functions, but not a convex and a concave function. The command `sum(f)` is equivalent to `f[0] + f[1] + ...+ f[len(f)-1]`.

Multiplication Scalar multiplication `a*f` of a piecewise-linear function `f` is defined if `a` is an `integer`, `float`, 1 by 1 `'d'` matrix. Matrix-matrix multiplications `a*f` or `f*a` are only defined if `a` is a dense or sparse 1 by 1 matrix.

Indexing and slicing Piecewise-linear functions admit single-argument indexing of the four types described in section 2.4. The result of an indexing or slicing operation is a new piecewise-linear function.

In the following example, `f` is the 1-norm of a vector variable `x` of length 10, `g` is its infinity-norm and `h` is the function

$$h(x) = \sum_k \phi(x[k]), \qquad \phi(u) = \begin{cases} 0 & |u| \leq 1 \\ |u| - 1 & 1 \leq |u| \leq 2 \\ 2|u| - 3 & |u| \geq 2. \end{cases}$$

```
>>> from cvxopt.modeling import variable, max
>>> x = variable(10, 'x')
>>> f = sum(abs(x))
>>> g = max(abs(x))
>>> h = sum(max(0, abs(x)-1, 2*abs(x)-3))
```

In-place operations If `f` is piecewise-linear then the in-place operations `f += u`, `f -= u`, `f *= u`, `f /= u` are defined if the corresponding expanded operations `f = f+u`, `f = f-u`, `f = f*u` and `f = f/u` are defined and if they do not change the length of `f`.

10.3 Constraints

Linear equality and inequality constraints of the form

$$f(x_1, \ldots, x_n) = 0, \qquad f(x_1, \ldots, x_n) \preceq 0,$$

where f is a convex function, are represented by `constraint` objects. Equality constraints are created by expressions of the form

```
f1 == f2.
```

Here `f1` and `f2` can be any objects for which the difference `f1-f2` yields an affine function. Inequality constraints are created by expressions of the form

```
f1 <= f2,    f2 >= f1,
```

where `f1` and `f2` can be any objects for which the difference `f1-f2` yields a convex piecewise-linear function. The comparison operators first convert the expressions to `f1-f2 == 0`, resp. `f1-f2 <= 0`, and then return a new `constraint` object with constraint function `f1-f2`.
In the following example we create three constraints

$$0 \preceq x \preceq \mathbf{1}, \qquad \mathbf{1}^T x = 2,$$

for a variable of length 5.

```
>>> x = variable(5,'x')
>>> c1 = (x <= 1)
>>> c2 = (x >= 0)
>>> c3 = (sum(x) == 2)
```

The built-in function `len()` returns the dimension of the constraint function.
Constraints have four public attributes.

type()

Returns `'='` if the constraint is an equality constraint, and `'<'` if the constraint is an inequality constraint.

value()

Returns the value of the constraint function.

multiplier

For a constraint `c`, `c.multiplier` is a `variable` object of dimension `len(c)`. It is used to represent the Lagrange multiplier or dual variable associated with the constraint. Its value is initialized as `None`, and can be modified by making an assignment to `c.multiplier.value`.

name

The name of the constraint. Changing the name of a constraint also changes the name of the multiplier of `c`. For example, the command `c.name = 'newname'` also changes `c.multiplier.name` to `'newname_mul'`.

10.4 Optimization Problems

Optimization problems are be constructed by calling the following function.

op([`objective`[, `constraints`[, `name`]]])

The first argument specifies the objective function to be minimized. It can be an affine or convex piecewise-linear function with length 1, a `variable` with length 1, or a scalar constant (`integer`, `float` or 1 by 1 dense `'d'` matrix). The default value is `0.0`.

The second argument is a single `constraint`, or a list of `constraint` objects. The default value is an empty list.

The third argument is a string with a name for the problem. The default value is the empty string.

The following attributes and methods are useful for examining and modifying optimization problems.

objective

The objective or cost function. One can write to this attribute to change the objective of an existing problem.

variables()

Returns a list of the variables of the problem.

constraints()

Returns a list of the constraints.

inequalities()

Returns a list of the inequality constraints.

equalities()

Returns a list of the equality constraints.

delconstraint(`c`)

Deletes constraint `c` from the problem.

addconstraint(`c`)

Adds constraint `c` to the problem.

An optimization problem with convex piecewise-linear objective and constraints can be solved by calling the method `solve()`.

solve([`format`[, `solver`]])

This function converts the optimization problem to a linear program in matrix form and then solves it using the solver described in section 8.3.

The first argument is either `'dense'` or `'sparse'`, and denotes the matrix types used in the matrix representation of the LP. The default value is `'dense'`.

The second argument is either `None`, `'glpk'` or `'mosek'`, and selects one of three available LP solvers: a default solver written in Python, the GLPK solver (if installed) or the MOSEK LP solver (if installed); see section 8.3. The default value is `None`.

The solver reports the outcome of optimization by setting the attribute `self.status` and by modifying the `value` attributes of the variables and the constraint multipliers of the problem.

- If the problem is solved to optimality, `self.status` is set to `'optimal'`. The `value` attributes of the variables in the problem are set to their computed solutions, and the `value` attributes of the multipliers of the constraints of the problem are set to the computed dual optimal solution.
- If it is determined that the problem is infeasible, `self.status` is set to `'primal infeasible'`. The `value` attributes of the variables are set to `None`. The `value` attributes of the multipliers of the constraints of the problem are set to a certificate of primal infeasibility. With the `'glpk'` option, `solve()` does not provide certificates of infeasibility.
- If it is determined that the problem is dual infeasible, `self.status` is set to `'dual infeasible'`. The `value` attributes of the multipliers of the constraints of the problem are set to `None`. The `value` attributes of the variables are set to a certificate of dual infeasibility. With the `'glpk'` option, `solve()` does not provide certificates of infeasibility.
- If the problem was not solved successfully, `self.status` is set to `'unknown'`. The `value` attributes of the variables and the constraint multipliers are set to `None`.

We refer to section 8.3 for details on the algorithms and the different solver options.
As an example we solve the LP

$$\begin{array}{ll} \text{minimize} & -4x - 5y \\ \text{subject to} & 2x + y \leq 3 \\ & x + 2y \leq 3 \\ & x \geq 0, \quad y \geq 0. \end{array}$$

```
>>> x = variable()
>>> y = variable()
>>> c1 = ( 2*x+y <= 3 )
>>> c2 = ( x+2*y <= 3 )
>>> c3 = ( x >= 0 )
>>> c4 = ( y >= 0 )
>>> lp1 = op(-4*x-5*y, [c1,c2,c3,c4])
>>> lp1.solve()
>>> lp1.status
'optimal'
>>> print lp1.objective.value()
[-9.00e+00]
>>> print x.value
[ 1.00e+00]
>>> print y.value
```

```
[ 1.00e+00]
>>> print c1.multiplier.value
[ 1.00e+00]
>>> print c2.multiplier.value
[ 2.00e+00]
>>> print c3.multiplier.value
[ 2.87e-08]
>>> print c4.multiplier.value
[ 2.80e-08]
```

We can solve the same LP in matrix form as follows.

```
>>> x = variable(2)
>>> A = matrix([[2.,1.,-1.,0.], [1.,2.,0.,-1.]])
>>> b = matrix([3.,3.,0.,0.])
>>> c = matrix([-4.,-5.])
>>> ineq = ( A*x <= b )
>>> lp2 = op(dot(c,x), ineq)
>>> lp2.solve()
>>> print lp2.objective.value()
[-9.00e+00]
>>> print x.value
[ 1.00e+00]
[ 1.00e+00]
>>> print ineq.multiplier.value
[1.00e+00]
[2.00e+00]
[2.87e-08]
[2.80e-08]
```

The `op` class also includes two methods for writing and reading files in MPS format[1].

tofile(`filename`)

If the problem is an LP, writes it to the file `'filename'` using the MPS format. Row and column labels are assigned based on the variable and constraint names in the LP.

fromfile(`filename`)

Reads the LP from the file `'filename'`. The file must be a fixed-format MPS file. Some features of the MPS format are not supported: comments beginning with dollar signs, the row types 'DE', 'DL', 'DG', and 'DN', and the capability of reading multiple righthand side, bound or range vectors.

10.5 Examples

Norm and Penalty Approximation In the first example we solve the norm approximation problems

$$\text{minimize} \quad \|Ax-b\|_\infty, \qquad \text{minimize} \quad \|Ax-b\|_1 \ ,$$

and the penalty approximation problem

$$\text{minimize} \quad \sum_k \phi((Ax-b)_k), \qquad \phi(u) = \begin{cases} 0 & |u| \leq 3/4 \\ |u| - 3/4 & 3/4 \leq |u| \leq 3/2 \\ 2|u| - 9/4 & |u| \geq 3/2. \end{cases}$$

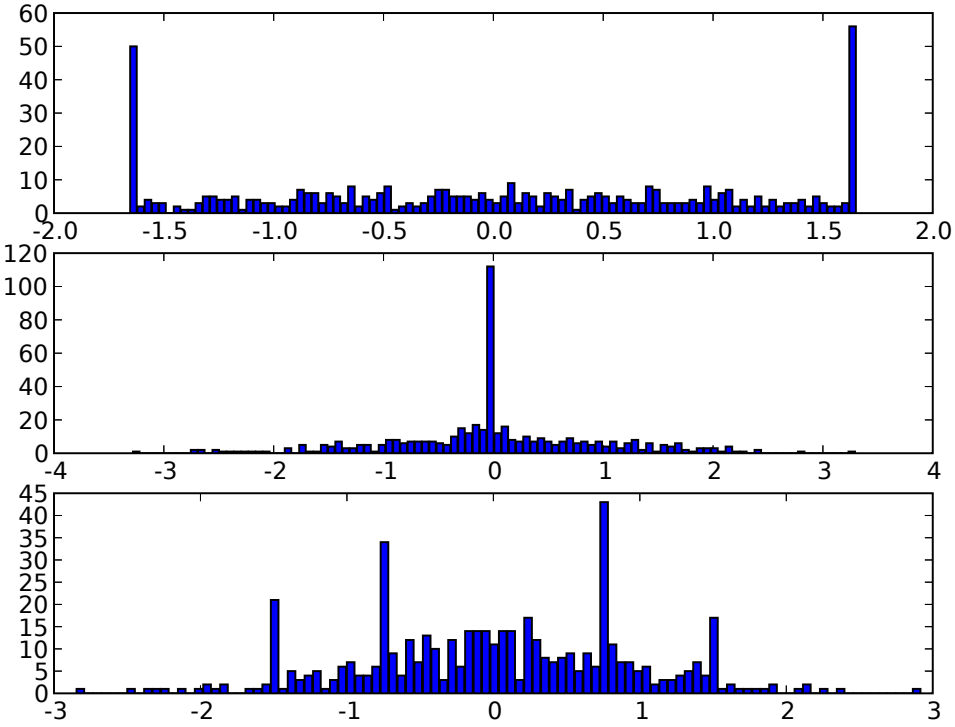
[1] `http://www-fp.mcs.anl.gov/otc/Guide/OptWeb/continuous/constrained/linearprog/mps.html`

We use randomly generated data.

The code uses the Matplotlib[2] package for plotting the histograms of the residual vectors for the two solutions. It generates the figure shown below.

```
from cvxopt.base import normal
from cvxopt.modeling import variable, op, max, sum
import pylab

m, n = 500, 100
A = normal(m,n)
b = normal(m)

x1 = variable(n)
op(max(abs(A*x1-b))).solve()

x2 = variable(n)
op(sum(abs(A*x2-b))).solve()

x3 = variable(n)
op(sum(max(0, abs(A*x3-b)-0.75, 2*abs(A*x3-b)-2.25))).solve()

pylab.subplot(311)
pylab.hist(A*x1.value-b, m/5)
pylab.subplot(312)
pylab.hist(A*x2.value-b, m/5)
pylab.subplot(313)
pylab.hist(A*x3.value-b, m/5)
pylab.show()
```

[2] http://matplotlib.sourceforge.net



Equivalently, we can formulate and solve the problems as LPs.

```
t = variable()
x1 = variable(n)
op(t, [-t <= A*x1-b, A*x1-b<=t]).solve()

u = variable(m)
x2 = variable(n)
op(sum(u), [-u <= A*x2+b, A*x2+b <= u]).solve()

v = variable(m)
x3 = variable(n)
op(sum(v), [v >= 0, v >= A*x3+b-0.75, v >= -(A*x3+b)-0.75, v >= 2*(A*x3-b)-2.25, v >=
```

Robust Linear Programming The robust LP

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & \sup_{\|v\|_\infty \leq 1} (a_i + v)^T x \leq b_i, \qquad i = 1, \ldots, m \end{array}$$

is equivalent to the problem

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & a_i^T x + \|x\|_1 \leq b_i, \qquad i = 1, \ldots, m. \end{array}$$

The following code computes the solution and the solution of the equivalent LP

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & a_i^T x + \mathbf{1}^T y \leq b_i, \qquad i = 1, \ldots, m \\ & -y \preceq x \preceq y \end{array}$$

for randomly generated data.

```
from cvxopt.base import normal, uniform
from cvxopt.modeling import variable, dot, op, sum

m, n = 500, 100
A = normal(m,n)
b = uniform(m)
c = normal(n)

x = variable(n)
op(dot(c,x), A*x+sum(abs(x)) <= b).solve()

x2 = variable(n)
y = variable(n)
op(dot(c,x2), [A*x2+sum(y) <= b, -y <= x2, x2 <= y]).solve()
```

1-Norm Support Vector Classifier The following problem arises in classification:

$$\begin{array}{ll} \text{minimize} & \|x\|_1 + \mathbf{1}^T u \\ \text{subject to} & Ax \succeq \mathbf{1} - u \\ & u \succeq 0. \end{array}$$

It can be solved as follows.

```
x = variable(A.size[1],'x')
u = variable(A.size[0],'u')
op(sum(abs(x)) + sum(u), [A*x >= 1-u, u >= 0]).solve()
```

An equivalent unconstrained formulation is

```
x = variable(A.size[1],'x')
op(sum(abs(x)) + sum(max(0,1-A*x))).solve()
```

Appendix A

C API

The API can be used to extend CVXOPT with interfaces to external C routines and libraries. A C program that creates or manipulates the dense or sparse matrix objects defined in `cvxopt.base` must include the `cvxopt.h` header file in the `src` directory of the distribution.
Before the C API can be used in an extension module it must be initialized by calling the macro `import_cvxopt`. As an example we show the module initialization from the `cvxopt.blas` module, which itself uses the API:

```
PyMODINIT_FUNC initblas(void)
{
  PyObject *m;

  m = Py_InitModule3("cvxopt.blas", blas_functions, blas__doc__);

  if (import_cvxopt() < 0)
    return;
}
```

A.1 Dense Matrices

As can be seen from the header file `cvxopt.h`, a `matrix` is essentially a structure with four fields. The fields `nrows` and `ncols` are two integers that specify the dimensions. The `id` field controls the type of the matrix and can have values `DOUBLE`, `INT` and `COMPLEX`. The `buffer` field is an array that contains the matrix elements stored contiguously in column-major order.
The following C functions can be used to create matrices.

`matrix *`**Matrix_New**`(int nrows, int ncols, int id)`

> Returns a `matrix` object of type `id` with `nrows` rows and `ncols` columns. The elements of the matrix are uninitialized.

`matrix *`**Matrix_NewFromMatrix**`(matrix *src, int id)`

> Returns a copy of the matrix `src` converted to type `id`. The following type conversions are allowed: `'i'` to `'d'`, `'i'` to `'z'` and `'d'` to `'z'`.

`matrix *`**Matrix_NewFromSequence**`(PyListObject *x, int id)`

> Creates a matrix of type `id` from the Python sequence type `x`. The returned matrix has size `(len(x),1)`. The size can be changed by modifying the `nrows` and `ncols` fields of the returned matrix.

To illustrate the creation and manipulation of dense matrices (as well as the Python C API), we show the code for the `uniform()` function from `cvxopt.random` described in section 2.7.

```
PyObject * uniform(PyObject *self, PyObject *args, PyObject *kwrds)
{
  matrix *obj;
  int i, nrows, ncols = 1;
  double a = 0, b = 1;
  char *kwlist[] = {"nrows", "ncols", "a", "b", NULL};

  if (!PyArg_ParseTupleAndKeywords(args, kwrds, "i|idd", kwlist,
          &nrows, &ncols, &a, &b)) return NULL;

  if ((nrows<0) || (ncols<0)) {
    PyErr_SetString(PyExc_TypeError, "dimensions must be non-negative");
    return NULL;
  }

  if (!(obj = Matrix_New(nrows, ncols, DOUBLE)))
    return PyErr_NoMemory();

  for (i = 0; i < nrows*ncols; i++)
    MAT_BUFD(obj)[i] = Uniform(a,b);

  return (PyObject *)obj;
}
```

A.2 Sparse Matrices

Sparse matrices are stored in compressed column storage (CCS) format. For a general `nrows` by `ncols` sparse matrix with `nnz` nonzero entries this means the following. The sparsity pattern and the nonzero values are stored in three fields:

values: A `'d'` or `'z'` matrix of size `(nnz,1)` with the nonzero entries of the matrix stored columnwise.

rowind: An array of integers of length `nnz` containing the row indices of the nonzero entries, stored in the same order as `values`.

colptr: An array of integers of length `ncols+1` with for each column of the matrix the index of the first element in `values` from that column. More precisely, `colptr[0]` is `0`, and for `k = 0, 1, ..., ncols-1`, `colptr[k+1]` is equal to `colptr[k]` plus the number of nonzeros in column k of the matrix. Thus, `colptr[ncols]` is equal to `nnz`, the number of nonzero entries.

For example, for the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 & 5 \\ 2 & 0 & 4 & 0 \\ 0 & 0 & 0 & 6 \\ 3 & 0 & 0 & 0 \end{bmatrix}$$

the elements of `values`, `rowind` and `colptr` are:

`values`: 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, `rowind`: 0, 1,3, 1, 0, 2, `colptr`: 0, 3, 3, 4, 6.

It is crucial that for each column the row indices in `rowind` are sorted; the equivalent representation

`values`: 3.0, 2.0, 1.0, 4.0, 5.0, 6.0, `rowind`: 3, 1, 0, 1, 0, 2, `colptr`: 0, 3, 3, 4, 6

is not allowed (and will likely cause the program to crash).
The `nzmax` field specifies the number of non-zero elements the matrix can store. It is equal to the length of `rowind` and `values`; this number can be larger that `colptr[nrows]`, but never less. This field makes it possible to preallocate a certain amount of memory to avoid reallocations if the matrix is constructed sequentially by filling in elements. In general the `nzmax` field can safely be ignored, however, since it will always be adjusted automatically as the number of non-zero elements grows.
The `id` field controls the type of the matrix and can have values `DOUBLE` and `COMPLEX`.
Sparse matrices are created using the following functions from the API.

`spmatrix *`**SpMatrix_New**`(int nrows, int ncols, int nzmax, int id)`

> Returns a sparse zero matrix with `nrows` rows and `ncols` columns. `nzmax` is the number of elements that will be allocated (the length of the `values` and `rowind` fields).

`spmatrix *`**SpMatrix_NewFromMatrix**`(spmatrix *src, int id)`

> Returns a copy the sparse matrix `src`.

`spmatrix *`**SpMatrix_NewFromIJV**`(matrix *I, matrix *J, matrix *V, int nrows, int ncols, int nzmax, int id)`

> Creates a sparse matrix with `nrows` rows and `ncols` columns from a triplet description. `I` and `J` must be integer matrices and `V` either a double or complex matrix, or `NULL`. If `V` is `NULL` the values of the entries in the matrix are undefined, otherwise they are specified by `V`. Repeated entries in `V` are summed. The number of allocated elements is given by `nzmax`, which is adjusted if it is smaller than the required amount.

We illustrate use of the sparse matrix class by listing the source code for the `real()` method, which returns the real part of a sparse matrix:

```
static PyObject * spmatrix_real(spmatrix *self) {

  if (SP_ID(self) != COMPLEX)
    return (PyObject *)SpMatrix_NewFromMatrix(self, 0, SP_ID(self));

  spmatrix *ret = SpMatrix_New(SP_NROWS(self), SP_NCOLS(self),
      SP_NNZ(self), DOUBLE);
  if (!ret) return PyErr_NoMemory();

  int i;
  for (i=0; i < SP_NNZ(self); i++)
    SP_VALD(ret)[i] = creal(SP_VALZ(self)[i]);

  memcpy(SP_COL(ret), SP_COL(self), (SP_NCOLS(self)+1)*sizeof(int_t));
  memcpy(SP_ROW(ret), SP_ROW(self), SP_NNZ(self)*sizeof(int_t));
  return (PyObject *)ret;
}
```

Appendix B

Matrix Formatting (`cvxopt.printing`)

This appendix describes ways to customize the formatting of CVXOPT matrices.
As with other Python objects, the functions `repr()` and `str()` return strings with printable representations of matrices. The command `"print A"` calls the function `str(A)`, whereas the command `"A"` calls `repr(A)`. The following example illustrates the default formatting of dense matrices.

```
>>> from cvxopt import base
>>> A = base.matrix(range(50), (5,10), 'd')
>>> A
<5x10 matrix, tc='d'>
>>> print A
[ 0.00e+00  5.00e+00  1.00e+01  1.50e+01  2.00e+01  2.50e+01  3.00e+01 ... ]
[ 1.00e+00  6.00e+00  1.10e+01  1.60e+01  2.10e+01  2.60e+01  3.10e+01 ... ]
[ 2.00e+00  7.00e+00  1.20e+01  1.70e+01  2.20e+01  2.70e+01  3.20e+01 ... ]
[ 3.00e+00  8.00e+00  1.30e+01  1.80e+01  2.30e+01  2.80e+01  3.30e+01 ... ]
[ 4.00e+00  9.00e+00  1.40e+01  1.90e+01  2.40e+01  2.90e+01  3.40e+01 ... ]
```

The format is parameterized by the dictionary `options` in the module `cvxopt.printing`. The parameters `options['iformat']` and `options['dformat']` determine, respectively, how integer and double/complex numbers are printed. The entries are Python format strings with default values `'% .2e'` for `'d'` and `'z'` matrices and `'% i'` for `'i'` matrices. The parameters `options['width']` and `options['height']` specify the maximum number of columns and rows that are shown. If `'width'` is set to a negative value, all columns are displayed. If `'height'` is set to a negative value, all rows are displayed. The default values of `'width'` and `'height'` are 7 and -1, respectively.

```
>>> from cvxopt import printing
>>> printing.options
{'width': 7, 'dformat': '% .2e', 'iformat': '% i', 'height': -1}
>>> printing.options['dformat'] = '%.1f'
>>> print_options['width'] = -1
>>> print A
[ 0.0  5.0 10.0 15.0 20.0 25.0 30.0 35.0 40.0 45.0]
[ 1.0  6.0 11.0 16.0 21.0 26.0 31.0 36.0 41.0 46.0]
[ 2.0  7.0 12.0 17.0 22.0 27.0 32.0 37.0 42.0 47.0]
[ 3.0  8.0 13.0 18.0 23.0 28.0 33.0 38.0 43.0 48.0]
[ 4.0  9.0 14.0 19.0 24.0 29.0 34.0 39.0 44.0 49.0]
```

In order to make the built-in Python functions `repr()` and `str()` accessible for further customization, two functions are provided in `cvxopt.base`. The function `base.matrix_repr()` is used when `repr()` is called with a matrix argument; and `base.matrix_str()` is used when `str()` is called with a matrix argument. By default,

the functions are set to `printing.matrix_repr_default()` and `printing.matrix_str_default()`, respectively, but they can be redefined to any other Python functions. For example, if we prefer `"A"` to return the same output as `"print A"`, we can simply redefine `base.matrix_repr()` as shown below.

```
>>> from cvxopt import base, printing
>>> A = base.matrix(range(4), (2,2), 'd')
>>> A
<2x2 matrix, tc='d'>
>>> base.matrix_repr = printing.matrix_str_default
>>> A
[ 0.00e+00  2.00e+00]
[ 1.00e+00  3.00e+00]
```

The formatting for sparse matrices is similar. The functions `repr()` and `str()` for sparse matrices are `base.spmatrix_repr()` and `base.spmatrix_str()`, respectively. By default, they are set to `printing.spmatrix_repr_default()` and `printing.spmatrix_repr_str()`.

```
>>> from cvxopt import base, printing
>>> A = base.spmatrix(range(5), range(5), range(5), (5,10) )
>>> A
<5x10 matrix, tc='d', nnz=5>
>>> print A
[ 0.00e+00     0         0         0         0         0         0     ... ]
[    0      1.00e+00     0         0         0         0         0     ... ]
[    0         0      2.00e+00     0         0         0         0     ... ]
[    0         0         0      3.00e+00     0         0         0     ... ]
[    0         0         0         0      4.00e+00     0         0     ... ]

>>> base.spmatrix_repr = printing.spmatrix_str_default
>>> A
[ 0.00e+00     0         0         0         0         0         0     ... ]
[    0      1.00e+00     0         0         0         0         0     ... ]
[    0         0      2.00e+00     0         0         0         0     ... ]
[    0         0         0      3.00e+00     0         0         0     ... ]
[    0         0         0         0      4.00e+00     0         0     ... ]
```

As can be seen from the example, the default behaviour is to print the entire matrix including structural zeros. An alternative triplet printing style is defined in `printing.spmatrix_str_triplet`.

```
>>> base.spmatrix_str = printing.spmatrix_str_triplet
>>> print A
(0,0)  0.00e+00
(1,1)  1.00e+00
(2,2)  2.00e+00
(3,3)  3.00e+00
(4,4)  4.00e+00
```