CVXOPT Documentation

Release 1.1.9

Martin S. Andersen, Joachim Dahl, and Lieven Vandenberghe

July 20, 2017

Contents

Release 1.1.9 – November 30, 2016

Martin Andersen, Joachim Dahl, and Lieven Vandenberghe

CHAPTER 1

Copyright and License

CHAPTER 2

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.

CVXOPT extends the built-in Python objects with two matrix objects: a `matrix` object for dense matrices and an `spmatrix` object for sparse matrices. These two matrix types are introduced in the chapter *Dense and Sparse Matrices*, together with the arithmetic operations and functions defined for them. The following chapters (*The BLAS Interface* and *Sparse Linear Equations*) describe interfaces to several libraries for dense and sparse matrix computations. The CVXOPT optimization routines are described in the chapters *Cone Programming* and *Modeling*. These include convex optimization solvers written in Python, interfaces to a few other optimization libraries, and a modeling tool for piecewise-linear convex optimization problems.

CVXOPT is organized in different modules.

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

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

`cvxopt.fftw` An optional interface to the discrete transform routines from FFTW (*Discrete Transforms*).

`cvxopt.amd` Interface to the approximate minimum degree ordering routine from AMD (*Matrix Orderings*).

`cvxopt.umfpack` Interface to the sparse LU solver from UMFPACK (*General Linear Equations*).

`cvxopt.cholmod` Interface to the sparse Cholesky solver from CHOLMOD (*Positive Definite Linear Equations*).

`cvxopt.solvers` Convex optimization routines and optional interfaces to solvers from GLPK, MOSEK, and DSDP5 (*Cone Programming* and *Nonlinear Convex Optimization*).

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

`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 3

Dense and Sparse Matrices

This chapter describes the two CVXOPT matrix types: *matrix* objects, used for dense matrix computations, and *spmatrix* objects, used for sparse matrix computations.

Dense Matrices

A dense matrix 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.

`cvxopt.`**`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 type code. 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 number), 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 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` 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(range(4), (2,2))
>>> from array import array
>>> A = matrix(array('i', [0,1,2,3]), (2,2))
>>> print(A)
[ 0  2]
[ 1  3]
```

In Python 2.7 the following also works.

```
>>> A = matrix(xrange(4), (2,2))
```

•If x is a dense or sparse matrix, 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]
```

NumPy arrays can be converted to matrices.

```
>>> 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 dense or sparse matrices and 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., if `x` is a list of lists, and has length one, then the argument `x` can be replaced by `x[0]`).

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

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

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

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.

Three functions are provided to create sparse matrices. The first, `spmatrix`, constructs a sparse matrix from a triplet description.

cvxopt.**spmatrix**(*x*, *I*, *J*[, *size*[, *tc*]])

I and J are sequences of integers (lists, tuples, array arrays, ...) 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 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.

```
>>> 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.

cvxopt.**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` objects) 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*). Numerical zeros are removed from the triplet description of the new matrix.

```
>>> from cvxopt import matrix, spmatrix, sparse
>>> A = matrix([[1., 2., 0.], [2., 1., 2.], [0., 2., 1.]])
>>> print(A)
[ 1.00e+00  2.00e+00  0.00e+00]
[ 2.00e+00  1.00e+00  2.00e+00]
[ 0.00e+00  2.00e+00  1.00e+00]
>>> B = spmatrix([], [], [], (3,3))
>>> print(B)
[0 0 0]
[0 0 0]
[0 0 0]
>>> C = spmatrix([3, 4, 5], [0, 1, 2], [0, 1, 2])
>>> print(C)
[ 3.00e+00     0          0    ]
[    0      4.00e+00      0    ]
[    0         0       5.00e+00]
>>> 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.

`cvxopt.`**`spdiag`**`(`*`x`*`)`

`x` is a dense or sparse matrix with a single row or column, or a list of square dense or sparse matrices or scalars. If `x` is a 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 elements in the list as its diagonal blocks.

```
>>> from cvxopt 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    ]
```

Arithmetic Operations

The following table lists the arithmetic operations defined for dense and sparse matrices. In the table `A` and `B` are dense or sparse matrices of compatible dimensions, `c` is a scalar (a Python number or a dense 1 by 1 matrix), `D` is a dense matrix, and `e` 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	`D % c`
Elementwise exponentiation	`D**e`

The type of the result of these operations generally follows the Python conventions. For example, if `A` and `c` are integer, then in Python 2 the division `A/c` is interpreted as integer division and results in a type `'i'` matrix, while in Python 3 it is interpreted as standard divison and results in a type `'d'` matrix. An exception to the Python conventions is elementwise exponentiation: if `D` is an integer matrix and `e` is an integer number than `D**e` is a matrix of type `'d'`.

Addition, subtraction, and matrix multiplication with two matrix operands result in a sparse matrix if both matrices are sparse, and in a dense matrix otherwise. The result of a scalar multiplication or division is dense if `A` is dense, and sparse if `A` is sparse. Postmultiplying a matrix with a number `c` means the same as premultiplying, i.e., scalar multiplication. Dividing a matrix by `c` means dividing all its entries by `c`.

If `c` in the expressions `A+c`, `c+A`, `A-c`, `c-A` is a number, then it is interpreted as a dense matrix with the same dimensions as `A`, type given by the type of `c`, and all its entries equal to `c`. If `c` is a 1 by 1 dense matrix and `A` is not 1 by 1, then `c` is interpreted as a dense matrix with the same size of `A` and all entries equal to `c[0]`.

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. If the product cannot be interpreted as a matrix-matrix product because the dimensions of `A` are incompatible, then the product is interpreted as the scalar multiplication with `c[0]`. 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]`.

The following in-place operations are also defined, but only if they do not change the type (sparse or dense, integer, real, or complex) of the matrix `A`. These in-place operations do not return a new matrix but modify the existing object `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 typecode `'i'`, then `A += B` is allowed if `B` has typecode `'i'`. It is not allowed if `B` has typecode `'d'` or `'z'` because the addition `A+B` results in a `'d'` or `'z'` matrix and therefore cannot be assigned to `A` without changing its type. As another example, if `A` is a sparse matrix, then `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 the scalar product `A *= c[0]`.)

In-place remainder is only defined for dense `A`.

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.

```
>>> 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]
```

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 import matrix, spmatrix
>>> 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]
>>> 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]
```

Expressions of the form `A[I]` or `A[I,J]` can also appear on the left-hand side of an assignment. The right-hand 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 right-hand side is a scalar, it is interpreted as a dense matrix with identical entries and the dimensions of the left-hand side. If the right-hand side is a sequence of numbers (list, tuple, `array` array, range object, ...) its values are interpreted as the coefficients of a dense matrix in column-major order. If the right-hand side is a matrix (`matrix` or `spmatrix`), it must have the same size as the left-hand side. Sparse matrices are converted to dense in the assignment to a dense matrix.

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 examples illustrate 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:] = range(4)
>>> print(A)
[ -1   1  -1   1]
[  1   6   9  13]
[ -2   6   0   2]
[  3   7   1   3]
>>> A = spmatrix([0,2,-1,2,-2,1], [0,1,2,0,2,1], [0,0,0,1,1,2])
>>> print(A)
[ 0.00e+00  2.00e+00     0    ]
[ 2.00e+00     0      1.00e+00]
[-1.00e+00 -2.00e+00     0    ]
>>> C = spmatrix([10,-20,30], [0,2,1], [0,0,1])
>>> print(C)
[ 1.00e+01     0    ]
[    0      3.00e+01]
[-2.00e+01     0    ]
```

```
>>> 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    ]
```

Attributes and Methods

Dense and sparse matrices have 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 character, 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.

In addition, sparse matrices have the following attributes.

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

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

CCS
A triplet (column pointers, row indices, 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.

The next example below illustrates assignments to `V`.

```
>>> from cvxopt 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    ]
>>> 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    ]
```

The following attributes and methods are defined for dense matrices.

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 examples.

```
>>> from cvxopt import matrix, spmatrix
>>> 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','wb')
>>> A.tofile(f)
>>> f.close()
>>> B = matrix(0.0, (2,3))
>>> f = open('mat.bin','rb')
>>> 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]
>>> A = spmatrix(range(5), [0,1,1,2,2], [0,0,1,1,2])
```

```
>>> f = open('test.bin','wb')
>>> A.V.tofile(f)
>>> A.I.tofile(f)
>>> A.J.tofile(f)
>>> f.close()
>>> f = open('test.bin','rb')
>>> 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 `dump` and `load` functions in the `pickle` module offer a convenient alternative for writing matrices to files and reading matrices from files.

Built-In Functions

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

len (*x*)
: If `x` is a dense matrix, returns the product of the number of rows and the number of columns. If `x` is a sparse matrix, returns the number of nonzero entries.

bool (*x*[])
: Returns `False` if `x` is a zero matrix and `True` otherwise.

max (*x*)
: If `x` is a dense matrix, returns the maximum element of `x`. If `x` is a sparse, returns the maximum nonzero element of `x`.

min (*x*)
: If `x` is a dense matrix, returns the minimum element of `x`. If `x` is a sparse matrix, returns the minimum nonzero 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`.

Dense and sparse matrices can be used as arguments to the `list`, `tuple`, `zip`, `map`, and `filter` functions described in the Python Library Reference. However, one should note that when used with sparse matrix arguments, these functions only consider the nonzero entries. For example, `list(A)` and `tuple(A)` construct a list, respectively a tuple, from the elements of `A` if `A` is dense, and of the nonzero elements of `A` if `A` is sparse.

`list(zip(A, B, ...))` returns a list of tuples, with the i-th tuple containing the i-th elements (or nonzero elements) of `A`, `B`, ...

```
>>> from cvxopt 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)
>>> list(zip(A, B))
[(-11.0, 0), (-5.0, 1), (-20.0, 2), (-6.0, 3), (0.0, 4), (7.0, 5)]
```

`list(map(f, A))`, where `f` is a function and `A` is a dense matrix, returns a list constructed by applying `f` to each element of `A`. If `A` is sparse, the function `f` is applied to each nonzero element of `A`. Multiple arguments can be provided, for example, as in `map(f, A, B)`, if `f` is a function with two arguments. In the following example, we return an integer 0-1 matrix with the result of an elementwise comparison.

```
>>> A = matrix([ [0.5, -0.1, 2.0], [1.5, 0.2, -0.1], [0.3, 1.0, 0.0]])
>>> print(A)
[ 5.00e-01  1.50e+00  3.00e-01]
[-1.00e-01  2.00e-01  1.00e+00]
[ 2.00e+00 -1.00e-01  0.00e+00]
>>> print(matrix(list(map(lambda x: 0 <= x <= 1, A)), A.size))
[ 1  0  1]
[ 0  1  1]
[ 0  0  1]
```

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

```
>>> A = matrix([[5, -4, 10, -7], [-1, -5, -6, 2], [6, 1, 5, 2],  [-1, 2, -3, -7]])
>>> print(A)
[  5  -1   6  -1]
[ -4  -5   1   2]
[ 10  -6   5  -3]
[ -7   2   2  -7]
>>> list(filter(lambda x: x%2, A))          # list of odd elements in A
[5, -7, -1, -5, 1, 5, -1, -3, -7]
>>> list(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` (or a nonzero element of `A` if `A` is sparse) is equal to `x` and `False` otherwise.

Other Matrix Functions

The following functions can be imported from CVXOPT.

`cvxopt.sqrt`(*x*)

The elementwise square root of a dense matrix `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.

As an example we take the elementwise square root of the sparse matrix

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

```
>>> from cvxopt import spmatrix, 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    ]
```

`cvxopt.sin`(*x*)
The sine function applied elementwise to a dense matrix `x`. The result is returned as a real matrix if `x` is an integer or real matrix and as a complex matrix otherwise.

`cvxopt.cos`(*x*)
The cosine function applied elementwise to a dense matrix `x`. The result is returned as a real matrix if `x` is an integer or real matrix and as a complex matrix otherwise.

`cvxopt.exp`(*x*)
The exponential function applied elementwise to a dense matrix `x`. The result is returned as a real matrix if `x` is an integer or real matrix and as a complex matrix otherwise.

`cvxopt.log`(*x*)
The natural logarithm applied elementwise to a dense matrix `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 nonpositive elements, or a complex matrix with zero elements.

`cvxopt.mul`(*x0*[, *x1*[, *x2* ...]])
If the arguments are dense or sparse matrices of the same size, returns the elementwise product of its arguments. The result is a sparse matrix if one or more of its arguments is sparse, and a dense matrix otherwise.

If the arguments include scalars, a scalar product with the scalar is made. (A 1 by 1 dense matrix is treated as a scalar if the dimensions of the other arguments are not all 1 by 1.)

`mul` can also be called with an iterable (list, tuple, range object, or generator) as its single argument, if the iterable generates a list of dense or sparse matrices or scalars.

```
>>> from cvxopt import matrix, spmatrix, mul
>>> A = matrix([[1.0, 2.0], [3.0, 4.0]])
>>> B = spmatrix([2.0, 3.0], [0, 1], [0, 1])
>>> print(mul(A, B, -1.0))
[-2.00e+00     0    ]
[    0     -1.20e+01]
>>> print(mul( matrix([k, k+1]) for k in [1,2,3] ))
[  6]
[ 24]
```

`cvxopt.div`(*x*, *y*)
Returns the elementwise division of `x` by `y`. `x` is a dense or sparse matrix, or a scalar (Python number of 1 by 1 dense matrix). `y` is a dense matrix or a scalar.

`cvxopt.max`(*x0*[, *x1*[, *x2* ...]])
When called with a single matrix argument, returns the maximum of the elements of the matrix (including the zero entries, if the matrix is sparse).

When called with multiple arguments, the arguments must be matrices of the same size, or scalars, and the elementwise maximum is returned. A 1 by 1 dense matrix is treated as a scalar if the other arguments are not all 1 by 1. If one of the arguments is scalar, and the other arguments are not all 1 by 1, then the scalar argument is interpreted as a dense matrix with all its entries equal to the scalar.

The result is a sparse matrix if all its arguments are sparse matrices. The result is a number if all its arguments are numbers. The result is a dense matrix if at least one of the arguments is a dense matrix.

max can also be called with an iterable (list, tuple, range object, or generator) as its single argument, if the iterable generates a list of dense or sparse matrices or scalars.

```
>>> from cvxopt import matrix, spmatrix, max
>>> A = spmatrix([2, -3], [0, 1], [0, 1])
>>> print(max(A, -A, 1))
[ 2.00e+00  1.00e+00]
[ 1.00e+00  3.00e+00]
```

It is important to note the difference between this *max* and the built-in *max*, explained in the previous section.

```
>>> from cvxopt import spmatrix
>>> A = spmatrix([-1.0, -2.0], [0,1], [0,1])
>>> max(A)          # built-in max of a sparse matrix takes maximum over nonzero␣
↪elements
-1.0
>>> max(A, -1.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotImplementedError: matrix comparison not implemented
>>> from cvxopt import max
>>> max(A)          # cvxopt.max takes maximum over all the  elements
0.0
>>> print(max(A, -1.5))
[-1.00e+00  0.00e+00]
[ 0.00e+00 -1.50e+00]
```

cvxopt.**min**(*x0*[, *x1*[, *x2* ...]])

When called with a single matrix argument, returns the minimum of the elements of the matrix (including the zero entries, if the matrix is sparse).

When called with multiple arguments, the arguments must be matrices of the same size, or scalars, and the elementwise maximum is returned. A 1 by 1 dense matrix is treated as a scalar if the other arguments are not all 1 by 1. If one of the arguments is scalar, and the other arguments are not all 1 by 1, then the scalar argument is interpreted as a dense matrix with all its entries equal to the scalar.

min can also be called with an iterable (list, tuple, range object, or generator) as its single argument, if the iterable generates a list of dense or sparse matrices or scalars.

Randomly Generated Matrices

The CVXOPT package 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) 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.

cvxopt.**normal**(*nrows*[, *ncols* = *1*[, *mean* = *0.0*[, *std* = *1.0*]]])

Returns a type `'d'` dense matrix of size `nrows` by `ncols` with elements chosen from a normal distribution

with mean `mean` and standard deviation `std`.

`cvxopt.`**`uniform`**`(`*nrows*[, *ncols* = *1*[, *a* = *0.0*[, *b* = *1.0*]]]`)`

Returns a type `'d'` dense matrix of size `nrows` by `ncols` matrix with elements uniformly distributed between `a` and `b`.

`cvxopt.`**`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)`.

`cvxopt.`**`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`.)

CHAPTER 4

The BLAS Interface

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 the section *Arithmetic Operations*, combined with the slicing and indexing of the section *Indexing and Slicing*. 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 the section *Arithmetic Operations* 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.

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

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

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

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 Hermitian matrices

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

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

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

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

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

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

$$\begin{bmatrix} 1 & X[0,1] & X[0,2] & \cdots & X[0,n-1] \\ 0 & 1 & X[1,2] & \cdots & X[1,n-1] \\ 0 & 0 & 1 & \cdots & X[2,n-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$ (uplo = 'U', 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 column k 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 k-th superdiagonal if `uplo` is `'U'`) and shifted horizontally so that the entries of the k-th 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

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

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

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`, with the same meaning as for symmetric band matrices. A `matrix X` of size $(k+1, n)$ can represent the band matrices

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

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

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`, with similar conventions

as for symmetric band matrices. A `matrix X` of size $(k+1, n)$ can represent the band matrices

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

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

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

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

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 can be viewed with the **`pydoc`** help program.

Level 1 BLAS

The level 1 functions implement vector operations.

`cvxopt.blas.`**`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.

`cvxopt.blas.`**`nrm2`**`(`*x*`)`

Euclidean norm of a vector: returns

$$\|x\|_2.$$

`cvxopt.blas.`**`asum`**`(`*x*`)`

1-Norm of a vector: returns

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

cvxopt.blas.**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.

cvxopt.blas.**swap**(*x*, *y*)
Interchanges two vectors:

$$x \leftrightarrow y.$$

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

cvxopt.blas.**copy**(*x*, *y*)
Copies a vector to another vector:

$$y := x.$$

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

cvxopt.blas.**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.

cvxopt.blas.**dot**(*x*, *y*)
Returns

$$x^H y.$$

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

cvxopt.blas.**dotu**(*x*, *y*)
Returns

$$x^T y.$$

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

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 the section *Matrix Classes*.

cvxopt.blas.**gemv**(*A*, *x*, *y*[, *trans = 'N'*, *alpha = 1.0*, *beta = 0.0*])
Matrix-vector product with a general matrix:

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

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.

cvxopt.blas.**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.

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

cvxopt.blas.**trmv**(*A*, *x*[, *uplo = 'L'*, *trans = 'N'*, *diag = 'N'*])

Matrix-vector product with a triangular matrix:

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

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

cvxopt.blas.**trsv**(*A*, *x*[, *uplo = 'L'*, *trans = 'N'*, *diag = 'N'*])

Solution of a nonsingular triangular set of linear equations:

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

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

cvxopt.blas.**gbmv**(*A*, *m*, *kl*, *x*, *y*[, *trans = 'N'*, *alpha = 1.0*, *beta = 0.0*])

Matrix-vector product with a general band matrix:

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

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

cvxopt.blas.**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`, `y` must have type `'d'`, and `alpha` and `beta` must be real.

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

cvxopt.blas.**tbmv**(*A*, *x*[, *uplo = 'L'*, *trans = 'N'*, *diag = 'N'*])
Matrix-vector product with a triangular band matrix:

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

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

cvxopt.blas.**tbsv**(*A*, *x*[, *uplo = 'L'*, *trans = 'N'*, *diag = 'N'*])
Solution of a triangular banded set of linear equations:

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

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'`).

cvxopt.blas.**ger**(*x*, *y*, *A*[, *alpha = 1.0*])
General rank-1 update:

$$A := A + \alpha x y^H,$$

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.

cvxopt.blas.**geru**(*x*, *y*, *A*[, *alpha = 1.0*])
General rank-1 update:

$$A := A + \alpha x y^T,$$

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.

cvxopt.blas.**syr**(*x*, *A*[, *uplo = 'L'*, *alpha = 1.0*])
Symmetric rank-1 update:

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

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

cvxopt.blas.**her**(*x*, *A*[, *uplo = 'L'*, *alpha = 1.0*])
Hermitian rank-1 update:

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

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.

cvxopt.blas.**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.

cvxopt.blas.**her2**(*x*, *y*, *A*[, *uplo = 'L'*, *alpha = 1.0*])

Symmetric rank-2 update:

$$A := A + \alpha x y^H + \bar{\alpha} y x^H,$$

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 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 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]
```

Level 3 BLAS

The level 3 BLAS include functions for matrix-matrix multiplication.

cvxopt.blas.**gemm**(*A*, *B*, *C*[, *transA = 'N'*, *transB = 'N'*, *alpha = 1.0*, *beta = 0.0*])

Matrix-matrix product of two general matrices:

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

where

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

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.

cvxopt.blas.**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:

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

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.

cvxopt.blas.**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:

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

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.

cvxopt.blas.**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:

$$B := \alpha\,\text{op}(A)B \quad (\text{side} = \text{'L'}),$$
$$B := \alpha B\,\text{op}(A) \quad (\text{side} = \text{'R'})$$

where

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

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

cvxopt.blas.**trsm**(*A*, *B*[, *side = 'L'*, *uplo = 'L'*, *transA = 'N'*, *diag = 'N'*, *alpha = 1.0*])

Solution of a nonsingular triangular system of equations:

$$B := \alpha\,\text{op}(A)^{-1}B \quad (\text{side} = \text{'L'}),$$
$$B := \alpha B\,\text{op}(A)^{-1} \quad (\text{side} = \text{'R'}),$$

where

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

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.

cvxopt.blas.**syrk**(*A*, *C*[, *uplo = 'L'*, *trans = 'N'*, *alpha = 1.0*, *beta = 0.0*])

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

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

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.

`cvxopt.blas.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:

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

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.

`cvxopt.blas.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:

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

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.

`cvxopt.blas.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:

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

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 5

The LAPACK Interface

The module `cvxopt.lapack` includes functions for solving dense sets of linear equations, for the corresponding matrix factorizations (LU, Cholesky,), for solving least-squares and least-norm problems, for QR factorization, for symmetric eigenvalue problems, singular value decomposition, and Schur factorization.

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 the section *Matrix Classes* 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

General Linear Equations

`cvxopt.lapack.`**`gesv`**(*A*, *B*[, *ipiv* = *None*])

Solves

$$AX = B,$$

where A and B are real or complex matrices, with A square and nonsingular.

The arguments `A` and `B` must have the same type (`'d'` or `'z'`). On entry, `B` contains the right-hand side B; on exit it contains the solution X. 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 the triangular factors in an LU factorization, 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`.

Raises an `ArithmeticError` if the matrix is singular.

cvxopt.lapack.**getrf**(*A*, *ipiv*)

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.

cvxopt.lapack.**getrs**(*A*, *ipiv*, *B*[, *trans = 'N'*])

Solves a general set of linear equations

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

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 entry, `B` contains the right-hand side B; on exit it contains the solution X. `B` must have the same type as `A`.

cvxopt.lapack.**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 matrix 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 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.

cvxopt.lapack.**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'`). On entry, `B` contains the right-hand side B; on exit it contains the solution X. 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 `A`, using the BLAS format for general band matrices (see the section *Matrix Classes*). 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 BLAS format for general band matrices. In this case, `gbsv` does not modify `A` and does not return the factorization.

Raises an `ArithmeticError` if the matrix is singular.

`cvxopt.lapack.`**`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 the section *Matrix Classes*), by providing the diagonals (stored as rows of a $k_u + k_l + 1$ by n matrix `A`), 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.

`cvxopt.lapack.`**`gbtrs`**(*{A*, *kl*, *ipiv*, *B*[, *trans* = *'N'*])

Solves a set of linear equations

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

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 entry, `B` contains the right-hand side B; on exit it contains the solution X. `B` must have the same type as `A`.

As an example, we solve a linear equation with

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

```
>>> from cvxopt 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, (n,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, (n,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, (n,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, (n,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, (n,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, with the diagonals stored in three separate vectors.

`cvxopt.lapack.`**`gtsv`**(*dl*, *d*, *du*, *B*)

Solves

$$AX = B,$$

where A is an n by n tridiagonal matrix.

The subdiagonal of A is stored as a matrix `dl` of length $n-1$, the diagonal is stored as a matrix `d` of length n, and the superdiagonal is 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. On entry, `B` contains the right-hand side B; on exit it contains the solution X.

Raises an `ArithmeticError` if the matrix is singular.

`cvxopt.lapack.`**`gttrf`**(*dl*, *d*, *du*, *du2*, *ipiv*)

LU factorization of an n by n tridiagonal matrix.

The subdiagonal of A is stored as a matrix `dl` of length $n-1$, the diagonal is stored as a matrix `d` of length n, and the superdiagonal is stored as a matrix `du` of length $n-1$. `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.

`cvxopt.lapack.`**`gttrs`**(*dl*, *d*, *du*, *du2*, *ipiv*, *B*[, *trans* = *'N'*])

Solves a set of linear equations

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

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 entry, `B` contains the right-hand side B; on exit it contains the solution X. `B` must have the same type as the other arguments.

Positive Definite Linear Equations

`cvxopt.lapack.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.

`cvxopt.lapack.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.

`cvxopt.lapack.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`.

`cvxopt.lapack.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 matrix inverse.

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

$$\begin{bmatrix} -\mathbf{diag}(d)^2 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{5.1}$$

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

```
>>> from cvxopt 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

$$A^T \mathbf{diag}(d)^{-2} A x_2 = b_2 + A^T \mathbf{diag}(d)^{-2} b_1$$
$$\mathbf{diag}(d)^2 x_1 = A x_2 - b_1.$$

```
>>> 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}*b1
>>> posv(B, x2)                                # x2 := B^{-1}*x2 = B^{-1}*(b2 + A^
↪T*diag(d)^{-2}*b1)
>>> gemv(Asc, x2, x1, beta=-1.0)               # x1 := Asc*x2 - x1 = diag(d)^{-1} *
↪(A*x2 - b1)
>>> x1 = div(x1, d)                            # x1 := diag(d)^{-1}*x1 = diag(d)^{-2} *
↪(A*x2 - b1)
```

There are separate routines for equations with positive definite band matrices.

`cvxopt.lapack.`**`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 *Matrix Classes*). 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.

`cvxopt.lapack.`**`pbtrf`**$(A[, uplo = 'L'])$

Cholesky factorization

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

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.

`cvxopt.lapack.`**`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.

`cvxopt.lapack.`**`ptsv`**(*d*, *e*, *B*)
Solves

$$AX = B,$$

where A is an n by n positive definite real symmetric or complex Hermitian tridiagonal matrix.

The diagonal of A 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

or

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.

`cvxopt.lapack.`**`pttrf`**(*d*, *e*)
or

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 containing the subdiagonal 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.

`cvxopt.lapack.`**`pttrs`**(*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

or

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`.

Symmetric and Hermitian Linear Equations

`cvxopt.lapack.`**`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.

`cvxopt.lapack.`**`sytrf`**(*A*, *ipiv*[, *uplo* = *'L'*])
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.

`cvxopt.lapack.`**`sytrs`**(*A*, *ipiv*, *B*[, *uplo* = *'L'*])
Solves

$$AX = B$$

given the

factorization computed by *sytrf* or *sysv*. `B` must have the same type as `A`.

`cvxopt.lapack.`**`sytri`**(*A*, *ipiv*[, *uplo* = *'L'*])
Computes the inverse of a real or complex symmetric matrix.

On entry, `A` and `ipiv` contain the

factorization computed by *sytrf* or *sysv*. On exit, `A` contains the inverse.

`cvxopt.lapack.`**`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.

`cvxopt.lapack.`**`hetrf`**(*A*, *ipiv*[, *uplo* = *'L'*])
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.

`cvxopt.lapack.`**`hetrs`**(*A*, *ipiv*, *B*[, *uplo* = *'L'*])
Solves

$$AX = B$$

given the

factorization computed by *hetrf* or *hesv*.

cvxopt.lapack.**hetri** (*A*, *ipiv*[, *uplo* = *'L'*])
Computes the inverse of a real symmetric or complex Hermitian matrix.

On entry, `A` and `ipiv` contain the

factorization computed by *hetrf* or *hesv*. On exit, `A` contains the inverse.

As an example we solve the KKT system (5.1).

```
>>> 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')
```

Triangular Linear Equations

cvxopt.lapack.**trtrs** (*A*, *B*[, *uplo* = *'L'*, *trans* = *'N'*, *diag* = *'N'*])
Solves a triangular set of equations

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

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).

cvxopt.lapack.**trtri** (*A*[, *uplo* = *'L'*, *diag* = *'N'*])
Computes the inverse of a real or complex triangular matrix A. On exit, `A` contains the inverse.

cvxopt.lapack.**tbtrs** (*A*, *B*[, *uplo* = *'L'*, *trans* = *'T'*, *diag* = *'N'*])
Solves a triangular set of equations

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

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.

Least-Squares and Least-Norm Problems

cvxopt.lapack.**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 matrix `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.

The following functions compute QR and LQ factorizations.

`cvxopt.lapack.`**`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 min{m, n}. On exit, R is stored in the upper triangular/trapezoidal 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`.

`cvxopt.lapack.`**`gelqf`**(*A*, *tau*)

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

$$A = LQ.$$

If A is m by n, then Q is n by n and orthogonal/unitary, and L is m by n and lower triangular (if m is less than or equal to n), or lower trapezoidal (if m is greater than or equal to n).

`tau` is a matrix of the same type as `A` and of length min{m, n}. On exit, L is stored in the lower triangular/trapezoidal part of `A`. The matrix Q is stored as a product of min{m, n} elementary reflectors in the first min{m, n} rows of `A` and in `tau`.

`cvxopt.lapack.`**`geqp3`**(*A*, *jpvt*, *tau*)

QR factorization with column pivoting of a real or complex matrix A:

$$AP = QR.$$

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 $\min\{m, n\}$. `jpvt` is an integer matrix of length n. On entry, if `jpvt[k]` is nonzero, then column k of A is permuted to the front of AP. Otherwise, column k is a free column.

On exit, `jpvt` contains the permutation P: the operation AP is equivalent to `A[:, jpvt-1]`. R is stored in the upper triangular/trapezoidal part of `A`. The matrix Q is stored as a product of $\min\{m, n\}$ elementary reflectors in the first min{m,:math:n} columns of `A` and in `tau`.

In most applications, the matrix Q is not needed explicitly, and it is sufficient to be able to make products with Q or its transpose. The functions *unmqr* and *ormqr* multiply a matrix with the orthogonal matrix computed by *geqrf*.

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

Product with a real orthogonal or complex unitary matrix:

$$\begin{aligned} C &:= \mathrm{op}(Q)C \quad (\mathrm{side} = \mathrm{'L'}),\\ C &:= C\,\mathrm{op}(Q) \quad (\mathrm{side} = \mathrm{'R'}), \end{aligned}$$

where

$$\mathrm{op}(Q) = \begin{cases} Q & \mathrm{trans} = \mathrm{'N'}\\ Q^T & \mathrm{trans} = \mathrm{'T'}\\ Q^H & \mathrm{trans} = \mathrm{'C'}. \end{cases}$$

If `A` is m by n, then Q is square of order m and orthogonal or unitary. Q is stored in the first $\min\{m, n\}$ columns of `A` and in `tau` as a product of $\min\{m, n\}$ elementary reflectors, as computed by *geqrf*. The matrices `A`, `tau`, and `C` must have the same type. `trans = 'T'` is only allowed if the typecode is `'d'`.

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

Identical to *unmqr* but works only for real matrices, and the possible values of `trans` are `'N'` and `'T'`.

As an 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 blas, lapack, matrix, normal
>>> m, n = 10, 5
>>> A, b = normal(m,n), normal(m,1)
>>> x1 = +b
>>> lapack.gels(+A, x1)                    # x1[:n] minimizes || A*x - 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]' * x2
>>> lapack.trtrs(A[:n,:], x2, uplo='U')    # x2[:n] := R1^{-1} * x2[:n]
>>> blas.nrm2(x1[:n] - x2[:n])
3.0050798580569307e-16
```

The next two functions make products with the orthogonal matrix computed by *gelqf*.

`cvxopt.lapack.`**`unmlq`**(*A*, *tau*, *C*[, *side* = *'L'*, *trans* = *'N'*])

Product with a real orthogonal or complex unitary matrix:

$$\begin{aligned} C &:= \mathrm{op}(Q)C \quad (\mathrm{side} = \mathrm{'L'}),\\ C &:= C\,\mathrm{op}(Q) \quad (\mathrm{side} = \mathrm{'R'}), \end{aligned}$$

where

$$\mathrm{op}(Q) = \begin{cases} Q & \mathrm{trans} = \mathrm{'N'}, \\ Q^T & \mathrm{trans} = \mathrm{'T'}, \\ Q^H & \mathrm{trans} = \mathrm{'C'}. \end{cases}$$

If `A` is m by n, then Q is square of order n and orthogonal or unitary. Q is stored in the first $\min\{m, n\}$ rows of `A` and in `tau` as a product of $\min\{m, n\}$ elementary reflectors, as computed by *gelqf*. The matrices `A`, `tau`, and `C` must have the same type. `trans = 'T'` is only allowed if the typecode is `'d'`.

`cvxopt.lapack.`**`ormlq`**(*A*, *tau*, *C*[, *side = 'L'*, *trans = 'N'*])
Identical to *unmlq* but works only for real matrices, and the possible values of `trans` or `'N'` and `'T'`.

As an example, we solve a least-norm problem by a direct call to *gels*, and by separate calls to *gelqf*, *ormlq*, and *trtrs*.

```
>>> from cvxopt import blas, lapack, matrix, normal
>>> m, n = 5, 10
>>> A, b = normal(m,n), normal(m,1)
>>> x1 = matrix(0.0, (n,1))
>>> x1[:m] = b
>>> lapack.gels(+A, x1)                    # x1 minimizes ||x||_2 subject to A*x = b
>>> tau = matrix(0.0, (m,1))
>>> lapack.gelqf(A, tau)                   # A = [L1, 0] * [Q1; Q2]
>>> x2 = matrix(0.0, (n,1))
>>> x2[:m] = b                             # x2 = [b; 0]
>>> lapack.trtrs(A[:,:m], x2)              # x2[:m] := L1^{-1} * x2[:m]
>>> lapack.ormlq(A, tau, x2, trans='T')    # x2 := [Q1, Q2]' * x2
>>> blas.nrm2(x1 - x2)
0.0
```

Finally, if the matrix Q is needed explicitly, it can be generated from the output of *geqrf* and *gelqf* using one of the following functions.

`cvxopt.lapack.`**`ungqr`**(*A*, *tau*)
If `A` has size m by n, and `tau` has length k, then, on entry, the first `k` columns of the matrix `A` and the entries of `tau` contai an unitary or orthogonal matrix Q of order m, as computed by *geqrf*. On exit, the first $\min\{m, n\}$ columns of Q are contained in the leading columns of `A`.

`cvxopt.lapack.`**`orgqr`**(*A*, *tau*)
Identical to *ungqr* but works only for real matrices.

`cvxopt.lapack.`**`unglq`**(*A*, *tau*)
If `A` has size m by n, and `tau` has length k, then, on entry, the first `k` rows of the matrix `A` and the entries of `tau` contain a unitary or orthogonal matrix Q of order n, as computed by *gelqf*. On exit, the first $\min\{m, n\}$ rows of Q are contained in the leading rows of `A`.

`cvxopt.lapack.`**`orglq`**(*A*, *tau*)
Identical to *unglq* but works only for real matrices.

We illustrate this with the QR factorization of the matrix

$$A = \begin{bmatrix} 6 & -5 & 4 \\ 6 & 3 & -4 \\ 19 & -2 & 7 \\ 6 & -10 & -5 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix}.$$

```
>>> from cvxopt import matrix, lapack
>>> A = matrix([ [6., 6., 19., 6.], [-5., 3., -2., -10.], [4., -4., 7., -5] ])
```

```
>>> m, n = A.size
>>> tau = matrix(0.0, (n,1))
>>> lapack.geqrf(A, tau)
>>> print(A[:n, :])              # Upper triangular part is R.
[-2.17e+01  5.08e+00 -4.76e+00]
[ 2.17e-01 -1.06e+01 -2.66e+00]
[ 6.87e-01  3.12e-01 -8.74e+00]
>>> Q1 = +A
>>> lapack.orgqr(Q1, tau)
>>> print(Q1)
[-2.77e-01  3.39e-01 -4.10e-01]
[-2.77e-01 -4.16e-01  7.35e-01]
[-8.77e-01 -2.32e-01 -2.53e-01]
[-2.77e-01  8.11e-01  4.76e-01]
>>> Q = matrix(0.0, (m,m))
>>> Q[:, :n] = A
>>> lapack.orgqr(Q, tau)
>>> print(Q)                     # Q = [ Q1, Q2]
[-2.77e-01  3.39e-01 -4.10e-01 -8.00e-01]
[-2.77e-01 -4.16e-01  7.35e-01 -4.58e-01]
[-8.77e-01 -2.32e-01 -2.53e-01  3.35e-01]
[-2.77e-01  8.11e-01  4.76e-01  1.96e-01]
```

The orthogonal matrix in the factorization

$$A = \begin{bmatrix} 3 & -16 & -10 & -1 \\ -2 & -12 & -3 & 4 \\ 9 & 19 & 6 & -6 \end{bmatrix} = Q \begin{bmatrix} R_1 & R_2 \end{bmatrix}$$

can be generated as follows.

```
>>> A = matrix([ [3., -2., 9.], [-16., -12., 19.], [-10., -3., 6.], [-1., 4., -6.] ])
>>> m, n = A.size
>>> tau = matrix(0.0, (m,1))
>>> lapack.geqrf(A, tau)
>>> R = +A
>>> print(R)                     # Upper trapezoidal part is [R1, R2].
[-9.70e+00 -1.52e+01 -3.09e+00  6.70e+00]
[-1.58e-01  2.30e+01  1.14e+01 -1.92e+00]
[ 7.09e-01 -5.57e-01  2.26e+00  2.09e+00]
>>> lapack.orgqr(A, tau)
>>> print(A[:, :m])              # Q is in the first m columns of A.
[-3.09e-01 -8.98e-01 -3.13e-01]
[ 2.06e-01 -3.85e-01  9.00e-01]
[-9.28e-01  2.14e-01  3.04e-01]
```

Symmetric and Hermitian Eigenvalue Decomposition

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

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

`cvxopt.lapack.`**`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.

`cvxopt.lapack.`**`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.

`cvxopt.lapack.`**`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 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.

`cvxopt.lapack.`**`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.

`cvxopt.lapack.`**`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.

`cvxopt.lapack.`**`heevd`**(*A*, *W*[, *jobz = 'N'*[, *uplo = 'L'*]])

This is an alternative to *heev*.

`cvxopt.lapack.`**`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.

The calling sequence is identical to *syevx*, except that `A` can be real or complex. `Z` must have the same type as `A`.

`cvxopt.lapack.`**`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*.

Generalized Symmetric Definite Eigenproblems

Three types of generalized eigenvalue problems can be solved:

$$\begin{aligned} AZ &= BZ\,\mathbf{diag}(\lambda) \quad \text{(type 1)}, \\ ABZ &= Z\,\mathbf{diag}(\lambda) \quad \text{(type 2)}, \\ BAZ &= Z\,\mathbf{diag}(\lambda) \quad \text{(type 3)}, \end{aligned} \tag{5.2}$$

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

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

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

Solves the generalized eigenproblem (5.2) 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.

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

Generalized eigenvalue problem (5.2) 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.

Singular Value Decomposition

`cvxopt.lapack.`**`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.

cvxopt.lapack.**gesdd**(*A*, *S*[, *jobz = 'N'*, *U = None*, *Vt = None*])

Singular value decomposition of a real or complex m by n matrix.. 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 :math‘m‘ is less than n) or a matrix of the same type as `A`.

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

Schur and Generalized Schur Factorization

cvxopt.lapack.**gees**(*A*[, *w = None*, *V = None*, *select = None*])

Computes the Schur factorization

$$A = VSV^T \quad (A \text{ real}), \qquad A = VSV^H \quad (A \text{ complex})$$

of a real or complex n by n matrix A.

If A is real, the matrix of Schur vectors V is orthogonal, and S is a real upper quasi-triangular matrix with 1 by 1 or 2 by 2 diagonal blocks. The 2 by 2 blocks correspond to complex conjugate pairs of eigenvalues of A. If A is complex, the matrix of Schur vectors V is unitary, and S is a complex upper triangular matrix with the eigenvalues of A on the diagonal.

The optional argument `w` is a complex matrix of length at least n. If it is provided, the eigenvalues of `A` are returned in `w`. The optional argument `V` is an n by n matrix of the same type as `A`. If it is provided, then the Schur vectors are returned in `V`.

The argument `select` is an optional ordering routine. It must be a Python function that can be called as `f(s)` with a complex argument `s`, and returns `True` or `False`. The eigenvalues for which `select` returns `True` will be selected to appear first along the diagonal. (In the real Schur factorization, if either one of a complex conjugate pair of eigenvalues is selected, then both are selected.)

On exit, `A` is replaced with the matrix S. The function `gees` returns an integer equal to the number of eigenvalues that were selected by the ordering routine. If `select` is `None`, then `gees` returns 0.

As an example we compute the complex Schur form of the matrix

$$A = \begin{bmatrix} -7 & -11 & -6 & -4 & 11 \\ 5 & -3 & 3 & -12 & 0 \\ 11 & 11 & -5 & -14 & 9 \\ -4 & 8 & 0 & 8 & 6 \\ 13 & -19 & -12 & -8 & 10 \end{bmatrix}.$$

```
>>> A = matrix([[-7., 5., 11., -4., 13.], [-11., -3., 11., 8., -19.], [-6., 3., -5.,␣
↪0., -12.],
                [-4., -12., -14., 8., -8.], [11., 0., 9., 6., 10.]])
>>> S = matrix(A, tc='z')
>>> w = matrix(0.0, (5,1), 'z')
>>> lapack.gees(S, w)
0
>>> print(S)
[ 5.67e+00+j1.69e+01 -2.13e+01+j2.85e+00  1.40e+00+j5.88e+00 -4.19e+00+j2.05e-01  3.
↪19e+00-j1.01e+01]
[ 0.00e+00-j0.00e+00  5.67e+00-j1.69e+01  1.09e+01+j5.93e-01 -3.29e+00-j1.26e+00 -1.
↪26e+01+j7.80e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  1.27e+01+j3.43e-17 -6.83e+00+j2.18e+00  5.
↪31e+00-j1.69e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00 -1.31e+01-j0.00e+00 -2.
↪60e-01-j0.00e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00 -7.
↪86e+00-j0.00e+00]
>>> print(w)
[ 5.67e+00+j1.69e+01]
[ 5.67e+00-j1.69e+01]
[ 1.27e+01+j3.43e-17]
[-1.31e+01-j0.00e+00]
[-7.86e+00-j0.00e+00]
```

An ordered Schur factorization with the eigenvalues in the left half of the complex plane ordered first, can be computed as follows.

```
>>> S = matrix(A, tc='z')
>>> def F(x): return (x.real < 0.0)
...
>>> lapack.gees(S, w, select = F)
2
>>> print(S)
[-1.31e+01-j0.00e+00 -1.72e-01+j7.93e-02 -2.81e+00+j1.46e+00  3.79e+00-j2.67e-01  5.
↪14e+00-j4.84e+00]
[ 0.00e+00-j0.00e+00 -7.86e+00-j0.00e+00 -1.43e+01+j8.31e+00  5.17e+00+j8.79e+00  2.
↪35e+00-j7.86e-01]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  5.67e+00+j1.69e+01 -1.71e+01-j1.41e+01  1.
↪83e+00-j4.63e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  5.67e+00-j1.69e+01 -8.
↪75e+00+j2.88e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  1.
↪27e+01+j3.43e-17]
>>> print(w)
[-1.31e+01-j0.00e+00]
[-7.86e+00-j0.00e+00]
[ 5.67e+00+j1.69e+01]
[ 5.67e+00-j1.69e+01]
[ 1.27e+01+j3.43e-17]
```

`cvxopt.lapack.gges`(*A*, *B*[, *a* = *None*, *b* = *None*, *Vl* = *None*, *Vr* = *None*, *select* = *None*])

Computes the generalized Schur factorization

$$A = V_l S V_r^T, \quad B = V_l T V_r^T \quad (A \text{ and } B \text{ real}),$$
$$A = V_l S V_r^H, \quad B = V_l T V_r^H, \quad (A \text{ and } B \text{ complex})$$

of a pair of real or complex n by n matrices A, B.

If A and B are real, then the matrices of left and right Schur vectors V_l and V_r are orthogonal, S is a real upper quasi-triangular matrix with 1 by 1 or 2 by 2 diagonal blocks, and T is a real triangular matrix with nonnegative diagonal. The 2 by 2 blocks along the diagonal of S correspond to complex conjugate pairs of generalized eigenvalues of A, B. If A and B are complex, the matrices of left and right Schur vectors V_l and V_r are unitary, S is complex upper triangular, and T is complex upper triangular with nonnegative real diagonal.

The optional arguments `a` and `b` are `'z'` and `'d'` matrices of length at least n. If these are provided, the generalized eigenvalues of `A`, `B` are returned in `a` and `b`. (The generalized eigenvalues are the ratios `a[k] / b[k]`.) The optional arguments `Vl` and `Vr` are n by n matrices of the same type as `A` and `B`. If they are provided, then the left Schur vectors are returned in `Vl` and the right Schur vectors are returned in `Vr`.

The argument `select` is an optional ordering routine. It must be a Python function that can be called as `f(x, y)` with a complex argument `x` and a real argument `y`, and returns `True` or `False`. The eigenvalues for which `select` returns `True` will be selected to appear first on the diagonal. (In the real Schur factorization, if either one of a complex conjugate pair of eigenvalues is selected, then both are selected.)

On exit, `A` is replaced with the matrix S and `B` is replaced with the matrix T. The function `gges` returns an integer equal to the number of eigenvalues that were selected by the ordering routine. If `select` is `None`, then `gges` returns 0.

As an example, we compute the generalized complex Schur form of the matrix A of the previous example, and

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

```
>>> A = matrix([[-7., 5., 11., -4., 13.], [-11., -3., 11., 8., -19.], [-6., 3., -5.,␣
↪0., -12.],
                [-4., -12., -14., 8., -8.], [11., 0., 9., 6., 10.]])
>>> B = matrix(0.0, (5,5))
>>> B[:19:6] = 1.0
>>> S = matrix(A, tc='z')
>>> T = matrix(B, tc='z')
>>> a = matrix(0.0, (5,1), 'z')
>>> b = matrix(0.0, (5,1))
>>> lapack.gges(S, T, a, b)
0
>>> print(S)
[ 6.64e+00-j8.87e+00 -7.81e+00-j7.53e+00  6.16e+00-j8.51e-01  1.18e+00+j9.17e+00  5.
↪88e+00-j4.51e+00]
[ 0.00e+00-j0.00e+00  8.48e+00+j1.13e+01 -2.12e-01+j1.00e+01  5.68e+00+j2.40e+00 -2.
↪47e+00+j9.38e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00 -1.39e+01-j0.00e+00  6.78e+00-j0.00e+00  1.
↪09e+01-j0.00e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00 -6.62e+00-j0.00e+00 -2.
↪28e-01-j0.00e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00 -2.
↪89e+01-j0.00e+00]
>>> print(T)
[ 6.46e-01-j0.00e+00  4.29e-01-j4.79e-02  2.02e-01-j3.71e-01  1.08e-01-j1.98e-01 -1.
↪95e-01+j3.58e-01]
[ 0.00e+00-j0.00e+00  8.25e-01-j0.00e+00 -2.17e-01+j3.11e-01 -1.16e-01+j1.67e-01  2.
↪10e-01-j3.01e-01]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  7.41e-01-j0.00e+00 -3.25e-01-j0.00e+00  5.
↪87e-01-j0.00e+00]
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  8.75e-01-j0.00e+00  4.
↪84e-01-j0.00e+00]
```

```
[ 0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.00e+00-j0.00e+00  0.
↪00e+00-j0.00e+00]
>>> print(a)
[ 6.64e+00-j8.87e+00]
[ 8.48e+00+j1.13e+01]
[-1.39e+01-j0.00e+00]
[-6.62e+00-j0.00e+00]
[-2.89e+01-j0.00e+00]
>>> print(b)
[ 6.46e-01]
[ 8.25e-01]
[ 7.41e-01]
[ 8.75e-01]
[ 0.00e+00]
```

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

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

(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 *blas.syrk* to form the Hessian matrix and the LAPACK function *posv* to solve the Newton system. The code can be further optimized by replacing the matrix-vector products with the level-2 BLAS function *blas.gemv*.

```
from cvxopt import matrix, log, mul, div, blas, lapack
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 6

Discrete Transforms

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

Discrete Fourier Transform

`cvxopt.fftw.dft(X)`

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

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

`cvxopt.fftw.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 module also includes a discrete *N*-dimensional Fourier transform. The input matrix is interpreted as an *N*-dimensional matrix stored in column-major order. The discrete *N*-dimensional Fourier transform computes the corresponding one-dimensional transform along each dimension. For example, the two-dimensional transform applies a one-dimensional transform to all the columns of the matrix, followed by a one-dimensional transform to all the rows of the matrix.

`cvxopt.fftw.dftn(X[, dims = X.size])`

Replaces a dense complex matrix with its *N*-dimensional discrete Fourier transform. The dimensions of the *N*-dimensional matrix are given by the *N*-tuple `dims`. The two-dimensional transform is computed as `dftn(X, X.size)`.

`cvxopt.fftw.idftn(X[, dims = X.size])`
Replaces a dense complex *N*-dimensional matrix with its inverse *N*-dimensional discrete Fourier transform. The dimensions of the matrix are given by the tuple `dims`. The two-dimensional inverse transform is computed as `idftn(X, X.size)`.

Discrete Cosine Transform

`cvxopt.fftw.dct(X[, type = 2])`
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).

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

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

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

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

`cvxopt.fftw.idct(X[, type = 2])`
Replaces the columns of a dense real matrix with the inverses of the discrete cosine transforms defined above.

The module also includes a discrete *N*-dimensional cosine transform. The input matrix is interpreted as an *N*-dimensional matrix stored in column-major order. The discrete *N*-dimensional cosine transform computes the corresponding one-dimensional transform along each dimension. For example, the two-dimensional transform applies a one-dimensional transform to all the rows of the matrix, followed by a one-dimensional transform to all the columns of the matrix.

`cvxopt.fftw.dctn(X[, dims = X.size, type = 2])`
Replaces a dense real matrix with its *N*-dimensional discrete cosine transform. The dimensions of the *N*-dimensional matrix are given by the *N*-tuple `dims`. The two-dimensional transform is computed as `dctn(X, X.size)`.

`cvxopt.fftw.idctn(X[, dims = X.size, type = 2])`
Replaces a dense real *N*-dimensional matrix with its inverse *N*-dimensional discrete cosine transform. The dimensions of the matrix are given by the tuple `dims`. The two-dimensional inverse transform is computed as `idctn(X, X.size)`.

Discrete Sine Transform

`cvxopt.fftw.dst(X, dims[, 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).

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

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

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

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

`cvxopt.fftw.`**`idst`**(*X*, *dims*[, *type = 1*])

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

The module also includes a discrete *N*-dimensional sine transform. The input matrix is interpreted as an *N*-dimensional matrix stored in column-major order. The discrete *N*-dimensional sine transform computes the corresponding one-dimensional transform along each dimension. For example, the two-dimensional transform applies a one-dimensional transform to all the rows of the matrix, followed by a one-dimensional transform to all the columns of the matrix.

`cvxopt.fftw.`**`dstn`**(*X*[, *dims = X.size*, *type = 2*])

Replaces a dense real matrix with its *N*-dimensional discrete sine transform. The dimensions of the *N*-dimensional matrix are given by the *N*-tuple `dims`. The two-dimensional transform is computed as `dstn(X, X.size)`.

`cvxopt.fftw.`**`idstn`**(*X*[, *dims = X.size*, *type = 2*])

Replaces a dense real *N*-dimensional matrix with its inverse *N*-dimensional discrete sine transform. The dimensions of the matrix are given by the tuple `dims`. The two-dimensional inverse transform is computed as `idstn(X, X.size)`.

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 right-hand sides of equations, are stored using the same conventions as in the BLAS and LAPACK modules.

Matrix Orderings

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

See also:

- 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.

`cvxopt.amd.`**`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 import spmatrix, amd
>>> A = spmatrix([10,3,5,-2,5,2], [0,2,1,2,2,3], [0,0,1,1,2,3])
>>> P = amd.order(A)
>>> print(P)
[ 1]
[ 0]
[ 2]
[ 3]
```

General Linear Equations

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:

- 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.

`cvxopt.umfpack.linsolve`(*A*, *B*[, *trans* = *'N'*])

Solves a sparse set of linear equations

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

where A is a sparse matrix and B is a dense matrix. The arguments `A` and `B` must have the same type (`'d'` or `'z'`) as `A`. On exit `B` contains the solution. Raises an `ArithmeticError` 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 import spmatrix, matrix, 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 *linsolve* is equivalent to the following three functions called in sequence.

cvxopt.umfpack.**symbolic**(*A*)
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 *numeric*.

cvxopt.umfpack.**numeric**(*A*, *F*)
Performs a numeric LU factorization of a sparse, possibly rectangular, matrix A. The argument F is the symbolic factorization computed by *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 *solve*. Raises an ArithmeticError if the matrix is singular.

cvxopt.umfpack.**solve**(*A*, *F*, *B*[, *trans* = *'N'*])
Solves a set of linear equations

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

where A is a sparse matrix and B is a dense matrix. The arguments A and B must have the same type. The argument F is a numeric factorization computed by *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 right-hand 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 import spmatrix, matrix, 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]
```

Positive Definite Linear Equations

`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

(or

factorizations of symmetric indefinite matrices (with L unit lower-triangular and D diagonal and nonsingular) if such a factorization exists.

See also:

- Y. Chen, T. A. Davis, W. W. Hager, S. Rajamanickam, Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate, ACM Transactions on Mathematical Software, 35(3), 22:1-22:14, 2008.

`cvxopt.cholmod.`**`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. See the comment on `options['nmethods']` for details on which ordering is used by CHOLMOD.

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 import matrix, spmatrix, 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]
```

`cvxopt.cholmod.`**`splinsolve`**(*A*, *B*[, *p* = *None*, *uplo* = *'L'*])

Similar to *`linsolve`* except that `B` is an *`spmatrix`* and that the solution is returned as an output argument (as a new `spmatrix`). `B` is not modified. See the comment on `options['nmethods']` for details on which ordering is used by CHOLMOD.

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*.

cvxopt.cholmod.**symbolic**(*A*[, *p* = *None*, *uplo* = *'L'*])

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.

The symbolic factorization is returned as an opaque C object that can be passed to *numeric*.

See the comment on `options['nmethods']` for details on which ordering is used by CHOLMOD.

cvxopt.cholmod.**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 *symbolic* applied to the matrix `A`, or to another sparse matrix with the same sparsity pattern and typecode, or by *numeric* applied to a matrix with the same sparsity pattern and typecode as `A`.

If `F` was created by a *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` equal to `'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.

cvxopt.cholmod.**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 *numeric*. `sys` is an integer with values between 0 and 8.

sys	equation
0	$AX = B$
1	$LDL^TX = B$
2	$LDX = B$
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.

cvxopt.cholmod.**spsolve**(*F*, *B*[, *sys* = *0*])

Similar to *solve*, except that `B` is a class:*spmatrix*, and the solution is returned as an output argument (as an `spmatrix`). `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]
```

`cvxopt.cholmod.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 factorization can be retrieved via *`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 the options `options['supernodal']` and `options['nmethods']` is summarized as follows (and described in detail in the CHOLMOD user guide).

`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['nmethods']` The default ordering used by the CHOLMOD is the ordering in the AMD library, but depending on the value of `options['nmethods']`. other orderings are also considered. If `nmethods` is equal to 2, the ordering specified by the user and the AMD ordering are compared, and the best of the two orderings is used. If the user does not specify an ordering, the AMD ordering is used. If equal to 1, the user must specify an ordering, and the ordering provided by the user is used. If equal to 0, all available orderings are compared and the best ordering is used. The available orderings include the AMD ordering, the ordering specified by the user (if any), and possibly other orderings if they are installed during the CHOLMOD installation. Default: 0.

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 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
```

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

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

where $\circ$ denotes Hadamard product.

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

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

         minimize    -log det 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 range(m) if I[k]==J[k]]  # position of diagonal elements

    # starting point: symmetric identity with nonzero pattern I,J
    K = spmatrix(0.0, I, J)
    K[::n+1] = 1.0
```

```
    # Kn is used in the line search
    Kn = spmatrix(0.0, 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 range(100):

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

        # compute Kinv by solving K*X = I
        Kinv[:] = 0.0
        Kinv[::n+1] = 1.0
        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 range(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

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 the sections *Linear Cone Programs* and *Quadratic Cone Programs*. The function `conelp` is restricted to problems with linear cost functions, and can detect primal and dual infeasibility. The function `coneqp` solves the general quadratic problem, but requires the problem to be strictly 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 the sections *Linear Programming*, *Quadratic Programming*, *Second-Order Cone Programming*, *Semidefinite Programming*. In the section *Exploiting Structure* 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.

Linear Cone Programs

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

Solves a pair of primal and dual cone programs

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

The primal variables are x and s. The dual variables are y, 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

$$\begin{aligned} C_0 &= \{u \in \mathbf{R}^l \mid u_k \geq 0,\ k = 1, \ldots, l\}, \\ C_{k+1} &= \{(u_0, u_1) \in \mathbf{R} \times \mathbf{R}^{r_k - 1} \mid u_0 \geq \|u_1\|_2\}, \quad k = 0, \ldots, M-1, \\ C_{k+M+1} &= \left\{\mathbf{vec}(u) \mid u \in \mathbf{S}_+^{t_k}\right\}, \quad k = 0, \ldots, N-1. \end{aligned}$$

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 the section *Exploiting Structure*.

`conelp` returns a dictionary that contains the result and information about the accuracy of the solution. The most important fields have 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 `'x'`, `'s'`, `'y'`, `'z'` 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

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

The other entries in the output dictionary summarize the accuracy with which these optimality conditions are satisfied. The fields `'primal objective'`, `'dual objective'`, and `'gap'` give the primal objective $c^T x$, dual objective $-h^T z - b^T y$, and the gap $s^T z$. The field `'relative gap'` is the relative gap, defined as

$$\frac{s^T z}{\max\{-c^T x, -h^T z - b^T y\}} \quad \text{if} \quad \max\{-c^T x, -h^T z - b^T y\} > 0$$

and `None` otherwise. The fields `'primal infeasibility'` and `'dual infeasibility'` are the residuals in the primal and dual equality constraints, defined as

$$\max\{\frac{\|Gx + s - h\|_2}{\max\{1, \|h\|_2\}}, \frac{\|Ax - b\|_2}{\max\{1, \|b\|_2\}}\}, \qquad \frac{\|G^T z + A^T y + c\|_2}{\max\{1, \|c\|_2\}},$$

respectively.

`'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

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

The field `'residual as primal infeasibility certificate'` gives the residual

$$\frac{\|G^T z + A^T y\|_2}{\max\{1, \|c\|_2\}}.$$

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

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

The field `'residual as dual infeasibility certificate'` gives the residual

$$\max\{\frac{\|Gx + s\|_2}{\max\{1, \|h\|_2\}}, \frac{\|Ax\|_2}{\max\{1, \|b\|_2\}}\}.$$

`'unknown'` This indicates that the algorithm terminated early due to numerical difficulties or because the maximum number of iterations was reached. The `'x'`, `'s'`, `'y'`, `'z'` entries contain the iterates when the algorithm terminated. Whether these entries are useful, as approximate solutions or certificates of primal and dual infeasibility, can be determined from the other fields in the dictionary.

The fields `'primal objective'`, `'dual objective'`, `'gap'`, `'relative gap'`, `'primal infeasibility'`, `'dual infeasibility'` are defined as when `'status'` is `'optimal'`. The field `'residual as primal infeasibility certificate'` is defined as

$$\frac{\|G^T z + A^T y\|_2}{-(h^T z + b^T y)\max\{1, \|h\|_2\}}.$$

if $h^T z + b^T y < 0$, and `None` otherwise. A small value of this residual indicates that y and z, divided by $-h^T z - b^T y$, are an approximate proof of primal infeasibility. The field `'residual as dual infeasibility certificate'` is defined as

$$\max\{\frac{\|Gx + s\|_2}{-c^T x \max\{1, \|h\|_2\}}, \frac{\|Ax\|_2}{-c^T x \max\{1, \|b\|_2\}}\}$$

if $c^T x < 0$, and as `None` otherwise. A small value indicates that x and s, divided by $-c^T x$ are an approximate proof of dual infeasibility.

It is required that

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

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

$$\begin{array}{ll} \text{minimize} & -6x_1 - 4x_2 - 5x_3 \\ \text{subject to} & 16x_1 - 14x_2 + 5x_3 \leq -3 \\ & 7x_1 + 2x_2 \leq 5 \\ & \left\| \begin{bmatrix} 8x_1 + 13x_2 - 12x_3 - 2 \\ -8x_1 + 18x_2 + 6x_3 - 14 \\ x_1 - 3x_2 - 17x_3 - 13 \end{bmatrix} \right\|_2 \leq -24x_1 - 7x_2 + 15x_3 + 12 \\ & \left\| \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \right\|_2 \leq 10 \\ & \begin{bmatrix} 7x_1 + 3x_2 + 9x_3 & -5x_1 + 13x_2 + 6x_3 & x_1 - 6x_2 - 6x_3 \\ -5x_1 + 13x_2 + 6x_3 & x_1 + 12x_2 - 7x_3 & -7x_1 - 10x_2 - 7x_3 \\ x_1 - 6x_2 - 6x_3 & -7x_1 - 10x_2 - 7x_3 & -4x_1 - 28x_2 - 11x_3 \end{bmatrix} \preceq \begin{bmatrix} 68 & -30 & -19 \\ -30 & 99 & 23 \\ -19 & 23 & 10 \end{bmatrix}. \end{array}$$

```
>>> from cvxopt import matrix, solvers
>>> c = matrix([-6., -4., -5.])
>>> G = matrix([[ 16., 7.,  24.,  -8.,   8.,  -1.,  0., -1.,  0.,  0.,
                   7.,  -5.,   1.,  -5.,   1.,  -7.,   1.,   -7.,  -4.],
                [-14., 2.,   7., -13., -18.,   3.,  0.,  0., -1.,  0.,
                   3.,  13.,  -6.,  13.,  12., -10.,  -6.,  -10., -28.],
                [  5., 0., -15.,  12.,  -6.,  17.,  0.,  0.,  0., -1.,
                   9.,   6.,  -6.,   6.,  -7.,  -7.,  -6.,   -7., -11.]])
>>> h = matrix( [ -3., 5.,  12.,  -2., -14., -13., 10.,  0.,  0.,  0.,
                  68., -30., -19., -30.,  99.,  23., -19.,   23.,  10.] )
>>> 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.,   1.,  -7.,  0.,  0.,  -4.],
                [-14., 2.,   7., -13., -18.,   3.,  0.,  0., -1.,  0.,
                   3.,  13.,  -6.,  0.,  12., -10.,  0.,  0., -28.],
                [  5., 0., -15.,  12.,  -6.,  17.,  0.,  0.,  0., -1.,
                   9.,   6.,  -6.,  0.,  -7.,  -7.,  0.,  0., -11.]])
>>> h = matrix( [ -3., 5.,  12.,  -2., -14., -13., 10.,  0.,  0.,  0.,
                  68., -30., -19.,  0.,  99.,  23.,  0.,  0.,  10.] )
```

Quadratic Cone Programs

`cvxopt.solvers.coneqp`(*P*, *q*[, *G*, *h*[, *dims*[, *A*, *b*[, *initvals*[, *kktsolver*]]]]])

Solves a pair of primal and dual quadratic cone programs

$$\begin{array}{ll} \text{minimize} & (1/2)x^T Px + q^T x \\ \text{subject to} & Gx + s = h \\ & Ax = b \\ & s \succeq 0 \end{array}$$

and

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

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

$$\begin{aligned} C_0 &= \{u \in \mathbf{R}^l \mid u_k \geq 0,\ k = 1, \ldots, l\}, \\ C_{k+1} &= \{(u_0, u_1) \in \mathbf{R} \times \mathbf{R}^{r_k - 1} \mid u_0 \geq \|u_1\|_2\}, \quad k = 0, \ldots, M-1, \\ C_{k+M+1} &= \left\{\mathbf{vec}(u) \mid u \in \mathbf{S}_+^{t_k}\right\}, \quad k = 0, \ldots, N-1. \end{aligned}$$

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 the section *Exploiting Structure*.

`coneqp` returns a dictionary that contains the result and information about the accuracy of the solution. The most important fields have 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

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

`'unknown'` This indicates that the algorithm terminated early due to numerical difficulties or because the maximum number of iterations was reached. The `'x'`, `'s'`, `'y'`, `'z'` entries contain the iterates when the algorithm terminated.

The other entries in the output dictionary summarize the accuracy with which the optimality conditions are satisfied. The fields `'primal objective'`, `'dual objective'`, and `'gap'` give the primal objective $c^T x$, the dual objective calculated as

$$(1/2)x^T P x + q^T x + z^T(Gx - h) + y^T(Ax - b)$$

and the gap $s^T z$. The field `'relative gap'` is the relative gap, defined as

$$\frac{s^T z}{-\text{primal objective}} \quad \text{if primal objective} < 0, \qquad \frac{s^T z}{\text{dual objective}} \quad \text{if dual objective} > 0,$$

and `None` otherwise. The fields `'primal infeasibility'` and `'dual infeasibility'` are the residuals in the primal and dual equality constraints, defined as

$$\max\{\frac{\|Gx + s - h\|_2}{\max\{1, \|h\|_2\}}, \frac{\|Ax - b\|_2}{\max\{1, \|b\|_2\}}\}, \qquad \frac{\|Px + G^T z + A^T y + q\|_2}{\max\{1, \|q\|_2\}},$$

respectively.

It is required that the problem is solvable and that

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

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 matrix, solvers
>>> 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]
```

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.

`cvxopt.solvers.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` to `'glpk'` or `'mosek'`; see the section *Optional Solvers*. The meaning of the other arguments and the return value are the same as for `conelp` called with `dims` equal to `{'l': G.size[0], 'q': [], 's': []}`.

The initial values are ignored when `solver` is `'mosek'` or `'glpk'`. With the GLPK option, the solver does not return certificates of primal or dual infeasibility: if the status is `'primal infeasible'` or `'dual infeasible'`, all entries of the output dictionary are `None`. If the GLPK or MOSEK solvers are used, and the code returns with status `'unknown'`, all the other fields in the output dictionary are `None`.

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 import matrix, 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]
```

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.

`cvxopt.solvers.qp`(*P*, *q*[, *G*, *h*[, *A*, *b*[, *solver*[, *initvals*]]]])

Solves the pair of primal and dual convex quadratic programs

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

and

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

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` to `'mosek'`; see the section *Optional Solvers*. The meaning of the other arguments and the return value is the same as for *coneqp* called with *dims* equal to `{'l': G.size[0], 'q': [], 's': []}`.

When `solver` is `'mosek'`, 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

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

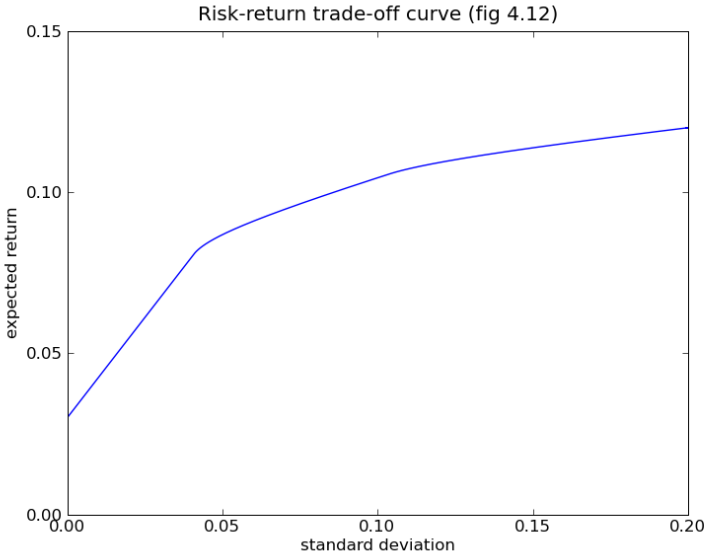
`'dual infeasible'` This means that a certificate of dual infeasibility has been found. The `'z'` and `'y'` entries are `None`, and the `'x'` and `'s'` entries are vectors that approximately satisfy

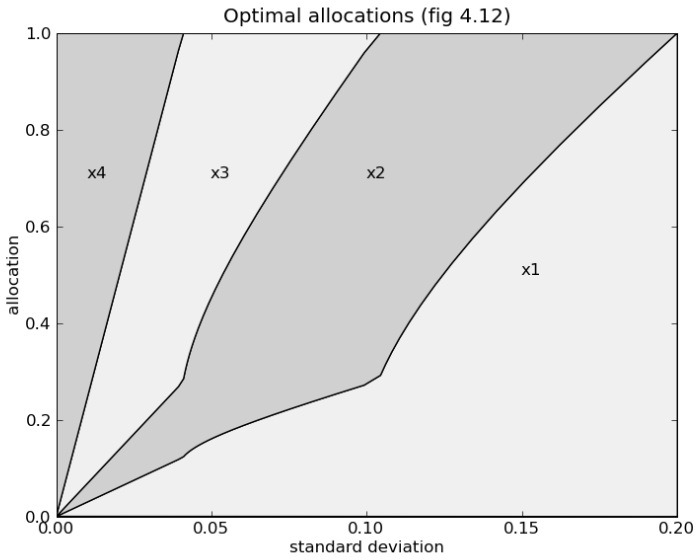
$$Px = 0, \qquad q^T x = -1, \qquad Gx + s = 0, \qquad Ax = 0, \qquad s \succeq 0.$$

As an example we compute the trade-off curve on page 187 of the book Convex Optimization, 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 μ. The code below computes the trade-off curve and produces two figures using the Matplotlib package.





```
from math import sqrt
from cvxopt 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 range(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()
```

Second-Order Cone Programming

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

cvxopt.solvers.**socp**(*c*[, *Gl*, *hl*[, *Gq*, *hq*[, *A*, *b*[, *solver*[, *primalstart*[, *dualstart*]]]]]])

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

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

and

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

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 right-hand 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, ..., 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` is `'mosek'`); see the section *Optional Solvers*. 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 that include entries with keys `'status'`, `'x'`, `'sl'`, `'sq'`, `'y'`, `'zl'`, `'zq'`. 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. The other entries in the output dictionary have the same meaning as in the output of *conelp*.

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 import matrix, 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]
```

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.

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

Solves the pair of primal and dual semidefinite programs

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

and

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

The inequalities

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

are componentwise vector inequalities. The other inequalities are matrix inequalities (ie, 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 right-hand 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` is `'dsdp'`); see the section *Optional Solvers*. 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 that includes entries 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. The other entries in the output dictionary have the same meaning as in the output of *`conelp`*.

We illustrate the calling sequence with a small example.

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

```
>>> from cvxopt import matrix, 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.,   8., 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.]]) ]
```

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 the sections *Linear Cone Programs* and *Quadratic Cone Programs*. Let

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

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

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:

$$W_1 = \mathbf{diag}\,(d), \qquad W_1^{-1} = \mathbf{diag}\,(d)^{-1}.$$

  This transformation is symmetric:

$$W_1^T = W_1.$$

- 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

$$\beta_k > 0, \qquad v_{k0} > 0, \qquad v_k^TJv_k = 1, \qquad J = \left[\begin{array}{cc} 1 & 0 \\ 0 & -I \end{array}\right].$$

  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 right-hand 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

$$\left[\begin{array}{ccc} P & A^T & G^TW^{-1} \\ A & 0 & 0 \\ G & 0 & -W^T \end{array}\right] \left[\begin{array}{c} \hat{u}_x \\ \hat{u}_y \\ \hat{u}_z \end{array}\right] = \left[\begin{array}{c} b_x \\ b_y \\ b_z \end{array}\right].$$

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, then `G(x, y[, alpha = 1.0, beta = 0.0, trans = 'N'])` should evaluate the matrix-vector products

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

- Similarly, if the argument `A` is a Python function, then `A(x, y[, alpha = 1.0, beta = 0.0, trans = 'N'])` should evaluate the matrix-vector products

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

- If the argument `P` of `coneqp` is a Python function, then `P(x, y[, alpha = 1.0, beta = 0.0])` should evaluate the matrix-vector products

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

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.) The code below takes advantage of this fact.

```
from cvxopt import blas, lapack, solvers, 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:]

        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:].^2.

        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+d2)

            # 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(di[:m],  u - x[n:] - z[:m])
            z[m:] = mul(di[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 blas, lapack, solvers, 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 import matrix, spdiag, mul, div, blas, lapack, solvers, sqrt
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:]), 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]
```

Optional Solvers

CVXOPT includes optional interfaces to several other optimization libraries.

GLPK *lp* with the `solver` option set to `'glpk'` uses the simplex algorithm in GLPK (GNU Linear Programming Kit).

MOSEK *lp*, *socp*, and *qp* with the `solver` option set to `'mosek'` option use MOSEK version 5.

DSDP *sdp* with the `solver` option set to `'dsdp'` uses the DSDP5.8.

GLPK, MOSEK and DSDP are not included in the CVXOPT distribution and need to be installed separately.

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

$$s \succeq 0, \qquad \frac{\|Gx + s - h\|_2}{\max\{1, \|h\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|Ax - b\|_2}{\max\{1, \|b\|_2\}} \leq \epsilon_{\text{feas}},$$

and

$$z \succeq 0, \qquad \frac{\|G^T z + A^T y + c\|_2}{\max\{1, \|c\|_2\}} \leq \epsilon_{\text{feas}},$$

and

$$s^T z \leq \epsilon_{\text{abs}} \qquad \text{or} \qquad \left(\min\left\{c^T x, h^T z + b^T y\right\} < 0 \quad \text{and} \quad \frac{s^T z}{-\min\{c^T x, h^T z + b^T y\}} \leq \epsilon_{\text{rel}} \right).$$

It returns with status `'primal infeasible'` if

$$z \succeq 0, \qquad \frac{\|G^T z + A^T y\|_2}{\max\{1, \|c\|_2\}} \leq \epsilon_{\text{feas}}, \qquad h^T z + b^T y = -1.$$

It returns with status `'dual infeasible'` if

$$s \succeq 0, \qquad \frac{\|Gx + s\|_2}{\max\{1, \|h\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|Ax\|_2}{\max\{1, \|b\|_2\}} \leq \epsilon_{\text{feas}}, \qquad c^T x = -1.$$

The functions `lp <cvxopt.solvers.lp`, *socp* and *sdp* call `conelp` and hence use the same stopping criteria.

The function *coneqp* terminates with status `'optimal'` if

$$s \succeq 0, \qquad \frac{\|Gx + s - h\|_2}{\max\{1, \|h\|_2\}} \leq \epsilon_{\text{feas}}, \qquad \frac{\|Ax - b\|_2}{\max\{1, \|b\|_2\}} \leq \epsilon_{\text{feas}},$$

and

$$z \succeq 0, \qquad \frac{\|Px + G^T z + A^T y + q\|_2}{\max\{1, \|q\|_2\}} \leq \epsilon_{\text{feas}},$$

and at least one of the following three conditions is satisfied:

$$s^T z \leq \epsilon_{\text{abs}}$$

or

$$\left(\frac{1}{2} x^T P x + q^T x < 0, \quad \text{and} \quad \frac{s^T z}{-(1/2) x^T P x - q^T x} \leq \epsilon_{\text{rel}} \right)$$

or

$$\left(L(x, y, z) > 0 \quad \text{and} \quad \frac{s^T z}{L(x, y, z)} \leq \epsilon_{\text{rel}} \right).$$

Here

$$L(x, y, z) = \frac{1}{2} x^T P x + q^T x + z^T (Gx - h) + y^T (Ax - b).$$

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['glpk']`. The entry must be a dictionary in which the key/value pairs are GLPK parameter names and values. For example, the command

```
>>> from cvxopt import solvers
>>> solvers.options['glpk'] = {'msg_lev' : 'GLP_MSG_OFF'}
```

turns off the screen output in subsequent *lp* calls 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 `mosek`. For details see Section 15 of the MOSEK Python API Manual.

For example, the commands

```
>>> from cvxopt import solvers
>>> import mosek
>>> solvers.options['mosek'] = {mosek.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['dsdp']` 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`).

It is also possible to override the options specified in the dictionary `solvers.options` by passing a dictionary with options as a keyword argument. For example, the commands

```
>>> from cvxopt import solvers
>>> opts = {'maxiters' : 50}
>>> solvers.conelp(c, G, h, options = opts)
```

override the options specified in the dictionary `solvers.options` and use the options in the dictionary `opts` instead. This is useful e.g. when several problem instances should be solved in parallel, but using different options.

CHAPTER 9

Nonlinear Convex Optimization

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 are convex and twice differentiable 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 the sections *Problems with Nonlinear Objectives* and *Problems with Linear Objectives*. A simpler interface for geometric programming problems is discussed in the section *Geometric Programming*. In the section *Exploiting Structure* 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.

Problems with Nonlinear Objectives

`cvxopt.solvers.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 x_0 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

$$\begin{aligned} C_0 &= \{u \in \mathbf{R}^l \mid u_k \geq 0,\ k = 1, \ldots, l\}, \\ C_{k+1} &= \{(u_0, u_1) \in \mathbf{R} \times \mathbf{R}^{r_k - 1} \mid u_0 \geq \|u_1\|_2\}, \quad k = 0, \ldots, M-1, \\ C_{k+M+1} &= \left\{\mathbf{vec}(u) \mid u \in \mathbf{S}_+^{t_k}\right\}, \quad k = 0, \ldots, N-1. \end{aligned}$$

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 the section *Exploiting Structure*.

`cp` returns a dictionary that contains the result and information about the accuracy of the solution. The most important fields have 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

$$\begin{gathered}\nabla f_0(x) + D\tilde{f}(x)^T z_{\mathrm{nl}} + G^T z_{\mathrm{l}} + A^T y = 0, \\ \tilde{f}(x) + s_{\mathrm{nl}} = 0, \quad k = 1, \ldots, m, \qquad Gx + s_{\mathrm{l}} = h, \qquad Ax = b, \\ s_{\mathrm{nl}} \succeq 0, \qquad s_{\mathrm{l}} \succeq 0, \qquad z_{\mathrm{nl}} \succeq 0, \qquad z_{\mathrm{l}} \succeq 0, \\ s_{\mathrm{nl}}^T z_{\mathrm{nl}} + s_{\mathrm{l}}^T z_{\mathrm{l}} = 0\end{gathered}$$

where $\tilde{f} = (f_1, \ldots, f_m)$.

'unknown' This indicates that the algorithm terminated before a solution was found, due to numerical difficulties or because the maximum number of iterations was reached. The 'x', 'snl', 'sl', 'y', 'znl', and 'zl' entries contain the iterates when the algorithm terminated.

cp solves the problem by applying *cpl* to the epigraph form problem

$$\begin{array}{ll}\text{minimize} & t \\ \text{subject to} & f_0(x) \leq t \\ & f_k(x) \leq 0, \quad k = 1, \ldots, m \\ & Gx \preceq h \\ & Ax = b.\end{array}$$

The other entries in the output dictionary of cp describe the accuracy of the solution and are copied from the output of *cpl* applied to this epigraph form problem.

cp requires that the problem is strictly primal and dual feasible and that

$$\mathbf{rank}(A) = p, \qquad \mathbf{rank}\left(\begin{bmatrix} \sum_{k=0}^m z_k \nabla^2 f_k(x) & A^T & \nabla f_1(x) & \cdots \nabla f_m(x) & G^T \end{bmatrix}\right) = n,$$

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, assuming it is solvable.

```
from cvxopt import solvers, 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

$$\text{minimize} \quad \sum_{k=1}^{m} \phi((Ax-b)_k), \qquad \phi(u) = \sqrt{\rho + u^2},$$

where $A \in \mathbf{R}^{m \times n}$.

```
from cvxopt import solvers, 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 import matrix, log, div, spdiag, 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]
```

Problems with Linear Objectives

`cvxopt.solvers.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 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

$$\begin{aligned} C_0 &= \{u \in \mathbf{R}^l \mid u_k \geq 0,\ k = 1, \ldots, l\}, \\ C_{k+1} &= \{(u_0, u_1) \in \mathbf{R} \times \mathbf{R}^{r_k - 1} \mid u_0 \geq \|u_1\|_2\}, \quad k = 0, \ldots, M-1, \\ C_{k+M+1} &= \left\{\mathbf{vec}(u) \mid u \in \mathbf{S}_+^{t_k}\right\}, \quad k = 0, \ldots, N-1. \end{aligned}$$

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 the section *Exploiting Structure*.

`cpl` returns a dictionary that contains the result and information about the accuracy of the solution. The most important fields have 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

$$\begin{aligned} c + Df(x)^T z_{\rm nl} + G^T z_{\rm l} + A^T y = 0, \\ f(x) + s_{\rm nl} = 0, \quad k = 1, \ldots, m, \qquad Gx + s_{\rm l} = h, \qquad Ax = b, \\ s_{\rm nl} \succeq 0, \qquad s_{\rm l} \succeq 0, \qquad z_{\rm nl} \succeq 0, \qquad z_{\rm l} \succeq 0, \\ s_{\rm nl}^T z_{\rm nl} + s_{\rm l}^T z_{\rm l} = 0. \end{aligned}$$

'unknown' This indicates that the algorithm terminated before a solution was found, due to numerical difficulties or because the maximum number of iterations was reached. The `'x'`, `'snl'`, `'sl'`, `'y'`, `'znl'`, and `'zl'` entries contain the iterates when the algorithm terminated.

The other entries in the output dictionary describe the accuracy of the solution. The entries `'primal objective'`, `'dual objective'`, `'gap'`, and `'relative gap'` give the primal objective $c^T x$, the dual objective, calculated as

$$c^T x + z_{\rm nl}^T f(x) + z_{\rm l}^T (Gx - h) + y^T (Ax - b),$$

the duality gap

$$s_{\rm nl}^T z_{\rm nl} + s_{\rm l}^T z_{\rm l},$$

and the relative gap. The relative gap is defined as

$$\frac{\text{gap}}{-\text{primal objective}} \quad \text{if primal objective} < 0, \qquad \frac{\text{gap}}{\text{dual objective}} \quad \text{if dual objective} > 0,$$

and `None` otherwise. The entry with key `'primal infeasibility'` gives the residual in the primal constraints,

$$\frac{\|(f(x) + s_{\rm nl}, Gx + s_{\rm l} - h, Ax - b)\|_2}{\max\{1, \|(f(x_0) + \mathbf{1}, Gx_0 + \mathbf{1} - h, Ax_0 - b)\|_2\}}$$

where x_0 is the point returned by `F()`. The entry with key `'dual infeasibility'` gives the residual

$$\frac{\|c + Df(x)^T z_{\rm nl} + G^T z_{\rm l} + A^T y\|_2}{\max\{1, \|c + Df(x_0)^T \mathbf{1} + G^T \mathbf{1}\|_2\}}.$$

`cpl` requires that the problem is strictly primal and dual feasible and that

$$\mathbf{rank}(A) = p, \qquad \mathbf{rank}\left(\begin{bmatrix} \sum_{k=0}^{m-1} z_k \nabla^2 f_k(x) & A^T & \nabla f_0(x) & \cdots \nabla f_{m-1}(x) & G^T \end{bmatrix}\right) = n,$$

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:

$$\begin{array}{ll} \text{minimize} & W + H \\ \text{subject to} & A_{\min,k}/h_k - w_k \leq 0, \quad k = 1, \ldots, 5 \\ & x_1 \geq 0, \quad x_2 \geq 0, \quad x_4 \geq 0 \\ & x_1 + w_1 + \rho \leq x_3, \quad x_2 + w_2 + \rho \leq x_3, \quad x_3 + w_3 + \rho \leq x_5, \\ & x_4 + w_4 + \rho \leq x_5, \quad x_5 + w_5 \leq W \\ & y_2 \geq 0, \quad y_3 \geq 0, \quad y_5 \geq 0 \\ & y_2 + h_2 + \rho \leq y_1, \quad y_1 + h_1 + \rho \leq y_4, y_3 + h_3 + \rho \leq y_4, \\ & y_4 + h_4 \leq H, \quad y_5 + h_5 \leq H \\ & h_k/\gamma \leq w_k \leq \gamma h_k, \quad k = 1, \ldots, 5. \end{array}$$

This problem has 22 variables

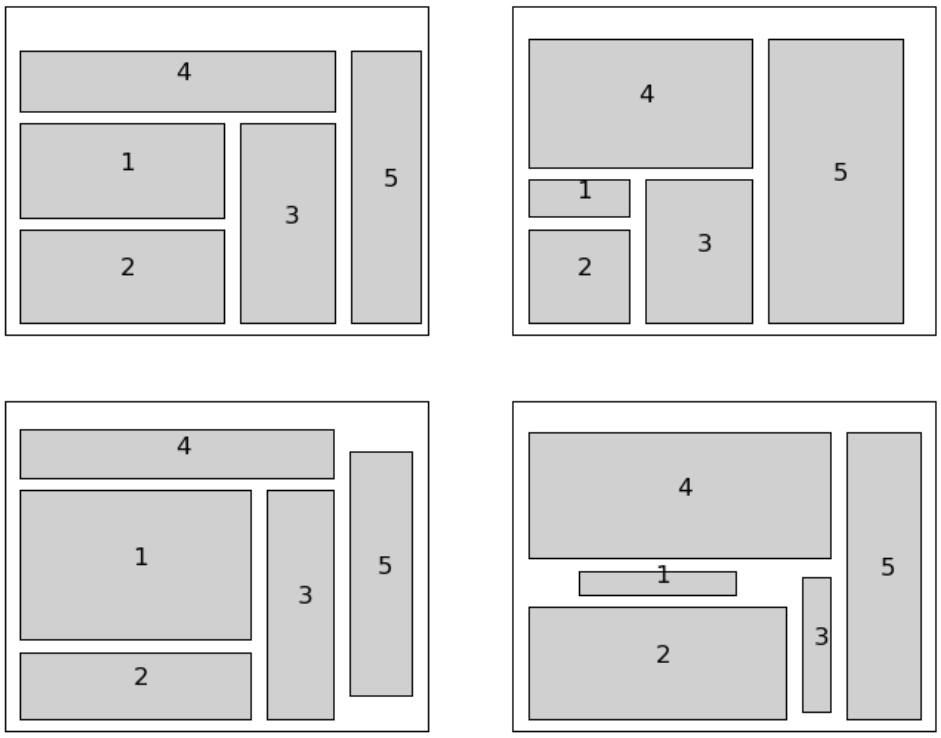
$$W, \qquad H, \qquad x \in \mathbf{R}^5, \qquad y \in \mathbf{R}^5, \qquad w \in \mathbf{R}^5, \qquad h \in \mathbf{R}^5,$$

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, 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
    #                 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], sol['x'][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 range(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 range(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 range(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 range(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()
```



Geometric Programming

`cvxopt.solvers.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

$$\mathbf{lse}(u) = \log \sum_k \exp(u_k), \qquad F = \left[\begin{array}{cccc} F_0^T & F_1^T & \cdots & F_m^T \end{array} \right]^T, \qquad g = \left[\begin{array}{cccc} g_0^T & g_1^T & \cdots & g_m^T \end{array} \right]^T,$$

and the vector inequality denotes componentwise inequality. `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

$$\begin{gathered}\nabla f_0(x) + \sum_{k=1}^{m} z_{\mathrm{nl},k} \nabla f_k(x) + G^T z_{\mathrm{l}} + A^T y = 0, \\ f_k(x) + s_{\mathrm{nl},k} = 0, \quad k = 1, \ldots, m \qquad Gx + s_{\mathrm{l}} = h, \qquad Ax = b, \\ s_{\mathrm{nl}} \succeq 0, \qquad s_{\mathrm{l}} \succeq 0, \qquad z_{\mathrm{nl}} \succeq 0, \qquad z_{\mathrm{l}} \succeq 0, \\ s_{\mathrm{nl}}^T z_{\mathrm{nl}} + s_{\mathrm{l}}^T z_{\mathrm{l}} = 0.\end{gathered}$$

`'unknown'` This indicates that the algorithm terminated before a solution was found, due to numerical difficulties or because the maximum number of iterations was reached. The `'x'`, `'snl'`, `'sl'`, `'y'`, `'znl'`, and `'zl'` contain the iterates when the algorithm terminated.

The other entries in the output dictionary describe the accuracy of the solution, and are taken from the output of *cp*.

As an example, we solve the small GP of section 2.4 of the paper A Tutorial on Geometric Programming. The posynomial form of the problem is

$$\begin{array}{ll}\text{minimize} & w^{-1}h^{-1}d^{-1} \\ \text{subject to} & (2/A_{\mathrm{wall}})hw + (2/A_{\mathrm{wall}})hd \leq 1 \\ & (1/A_{\mathrm{flr}})wd \leq 1 \\ & \alpha w h^{-1} \leq 1 \\ & (1/\beta)hw^{-1} \leq 1 \\ & \gamma w d^{-1} \leq 1 \\ & (1/\delta)dw^{-1} \leq 1\end{array}$$

with variables h, w, d.

```
from cvxopt import matrix, log, exp, 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'] )
```

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

$$\begin{bmatrix} H & A^T & \tilde{G}^T \\ A & 0 & 0 \\ \tilde{G} & 0 & -W^T W \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}, \tag{9.2}$$

where

$$H = \sum_{k=0}^{m} z_k \nabla^2 f_k(x), \qquad \tilde{G} = \left[\begin{array}{cccc} \nabla f_1(x) & \cdots & \nabla f_m(x) & G^T \end{array} \right]^T.$$

The matrix W depends on the current iterates and is defined as follows. Suppose

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

where

$$u_{\mathrm{nl}} \in \mathbf{R}^m, \qquad u_{\mathrm{l}} \in \mathbf{R}^l, \qquad u_{\mathrm{q},k} \in \mathbf{R}^{r_k}, \quad k = 0, \ldots, M-1, \qquad u_{\mathrm{s},k} \in \mathbf{S}^{t_k}, \quad k = 0, \ldots, N-1.$$

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*:

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

  where

  $$\beta_k > 0, \qquad v_{k0} > 0, \qquad v_k^T J v_k = 1, \qquad J = \left[\begin{array}{cc} 1 & 0 \\ 0 & -I \end{array} \right].$$

  These transformations are also symmetric:

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

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

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

  In general, this operation is not symmetric, and

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

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 right-hand side. On exit, they should contain the solution of the KKT system, with the last component scaled, i.e., on exit,

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

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 `cp` is a Python function, then `G(u, v[, alpha = 1.0, beta = 0.0, trans = 'N'])` should evaluates the matrix-vector products

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

- Similarly, if the argument `A` is a Python function, then `A(u, v[, alpha = 1.0, beta = 0.0, trans = 'N'])` should evaluate the matrix-vector products

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

- 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, then `Df(u, v[, alpha = 1.0, beta = 0.0, trans = 'N'])` should evaluate the matrix-vector products

$$v := \alpha Df(x) u + \beta v \quad (\text{trans} = \text{'N'}), \qquad v := \alpha Df(x)^T u + \beta v \quad (\text{trans} = \text{'T'}).$$

  If `H` is a Python function, then `H(u, v[, alpha, beta])` should evaluate the matrix-vector product

$$v := \alpha H u + \beta v.$$

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 import matrix, spdiag, mul, div, log, 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))
    v = matrix(0.0, (m,1))
    def Fkkt(x, z, W):
        ds = (2.0 * div(1 + x**2, (1 - x**2)**2))**-0.5
        Asc = A * spdiag(ds)
        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']
```

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 *cpl*.

`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* and *gp* call `cpl` and hence use the same stopping criteria (with $x_0 = 0$ for `gp`).

CHAPTER 10

Modeling

The module `cvxopt.modeling` can be used to specify and solve optimization problems with convex piecewise-linear objective and constraint functions. Using this modeling tool, one can specify an optimization problem by first defining the optimization variables (see the section *Variables*), and then specifying 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 the section *Functions*).

A more general Python convex modeling package is CVXPY.

Variables

Optimization variables are represented by `variable` objects.

`cvxopt.modeling.`**`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 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]
```

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 the section *Arithmetic Operations*, 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 the section *Arithmetic Operations*, 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]` equal to `len(v)`. The operation `v * a` is defined if `a` is scalar, or if `len(v)` is 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.

`cvxopt.modeling.`**`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`.

`cvxopt.modeling.`**`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` is equivalent to the function *`blas.dot`*, 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 the section *Indexing and Slicing*. 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 the section *Indexing and Slicing*. 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`.

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'`.

Optimization Problems

Optimization problems are be constructed by calling the following function.

`cvxopt.modeling.`**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.

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 the section *Linear Programming*.

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: the default solver written in Python, the GLPK solver (if installed) or the MOSEK LP solver (if installed); see the section *Linear Programming*. 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 the section *Linear Programming* for details on the algorithms and the different solver options.

As an example we solve the LP

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

```
>>> from cvxopt.modeling import op
>>> 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.

```
>>> from cvxopt.modeling import op, dot
>>> 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.

`tofile(filename) :noindex:`
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) :noindex:`
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.

Examples

Norm and Penalty Approximation

In the first example we solve the norm approximation problems

$$\text{minimize} \quad \|Ax-b\|_\infty, \qquad \text{minimize} \quad \|Ax-b\|_1 \ ,$$

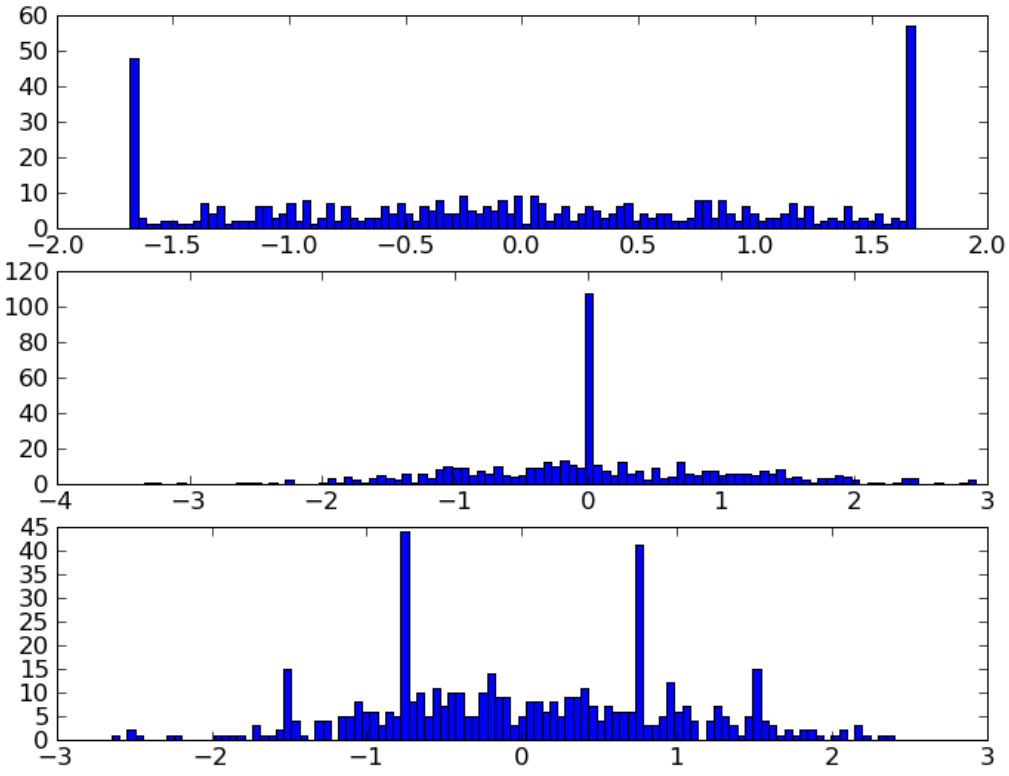
and the penalty approximation problem

$$\text{minimize} \quad \sum_k \phi((Ax-b)_k), \qquad \phi(u) = \begin{cases} 0 & |u| \leq 3/4 \\ |u| - 3/4 & 3/4 \leq |u| \leq 3/2 \\ 2|u| - 9/4 & |u| \geq 3/2. \end{cases}$$

We use randomly generated data.

The code uses the Matplotlib package for plotting the histograms of the residual vectors for the two solutions. It generates the figure shown below.

```
from cvxopt 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()
```



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 >= -2*(A*x3-b)-2.25]).solve()
```

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 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()
```

CHAPTER 11

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 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:

```
#if PY_MAJOR_VERSION >= 3

static PyModuleDef blas_module = {
    PyModuleDef_HEAD_INIT,
    "blas",
    blas__doc__,
    -1,
    blas_functions,
    NULL, NULL, NULL, NULL
};

PyMODINIT_FUNC PyInit_blas(void)
{
  PyObject *m;
  if (!(m = PyModule_Create(&blas_module))) return NULL;
  if (import_cvxopt() < 0) return NULL;
  return m;
}

#else

PyMODINIT_FUNC initblas(void)
{
  PyObject *m;
  m = Py_InitModule3("cvxopt.blas", blas_functions, blas__doc__);
  if (import_cvxopt() < 0) return ;
}
```

```
#endif
```

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 *`cvxopt.uniform`* function described in the section *Randomly Generated Matrices*.

```
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;
}
```

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 var{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;
}
```

CHAPTER 12

Matrix Formatting

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`' executes '`str(A)`', whereas the command '`A`' calls '`repr(A)`'. The following example illustrates the default formatting of dense matrices.

```
>>> from cvxopt import matrix
>>> A = 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 `options['width']` is set to a negative value, all columns are displayed. If `options['height']` is set to a negative value, all rows are displayed. The default values of `options['width']` and `options['height']` are 7 and -1, respectively.

```
>>> from cvxopt import printing
>>> printing.options
{'width': 7, 'dformat': '% .2e', 'iformat': '% i', 'height': -1}
>>> printing.options['dformat'] = '%.1f'
>>> printing.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. The function `cvxopt.matrix_repr` is used when `repr` is called with a matrix argument; and `cvxopt.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 `cvxopt.matrix_repr` as shown below.

```
>>> import cvxopt
>>> from cvxopt import matrix, printing
>>> A = matrix(range(4), (2,2), 'd')
>>> A
<2x2 matrix, tc='d'>
>>> cvxopt.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 `cvxopt.spmatrix_repr` and `cvxopt.spmatrix_str`, respectively. By default, they are set to `printing.spmatrix_repr_default` and `printing.spmatrix_repr_str`.

```
>>> import cvxopt
>>> from cvxopt import printing, spmatrix
>>> A = spmatrix(range(5), range(5), range(5), (5,10))
>>> A
<5x10 sparse 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      ... ]
```

```
>>> cvxopt.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`.

```
>>> cvxopt.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
```

Index

D

E

F

I

J

L

M

N

O

R

S

T

V