MOSEK Fusion for Java
Version 7.1 (rev. 27)

www.mosek.com

ii

- Published by MOSEK ApS, Denmark.

Contents

Chapter 1

Contact information

Phone	+45 3917 9907	
Fax	+45 3917 9823	
WEB	http://www.mosek.com	
Email	sales@mosek.com	Sales, pricing, and licensing.
	support@mosek.com	Technical support, questions and bug reports.
	info@mosek.com	Everything else.
Mail	MOSEK ApS	
	Fruebjergvej 3, Box 16	
	2100 Copenhagen Ō	
	Denmark	

License agreement

Before using the MOSEK software, please read the license agreement available in the distribution at

`mosek\7\license.pdf`

Chapter 2

Introduction

The most widespread class of optimization problems is *linear optimization problems*, where all relations are linear. The tremendous success of both applications and theory of linear optimization can be ascribed to the following factors:

- The data required are simple, i.e., just matrices and vectors.

- Convexity is guaranteed since the problem is convex by construction.

- Linear functions are trivially differentiable.

- There exist very efficient algorithms and software for solving linear problems.

- Duality properties for linear optimization are nice and simple.

Even if the linear optimization model is only an approximation to the true problem at hand, then the many advantages of linear optimization may outweight the disadvantages. In some cases, however, the problem formulation is inherently nonlinear and a linear approximation is either intractable or inadequate. *Conic optimization* has proved to be a very expressive and powerful way to introduce nonlinearities, while preserving all the nice properties of linear optimization listed above.

The fundamental expression in linear optimization is a linear expression of the form

$$Ax - b \in K$$

where $K = \{y : y \geq 0\}$, i.e.,

$$Ax - b \geq 0.$$

In conic optimization a wider class of convex sets K are allowed, for example in 3 dimensions K may correspond to an ice cream cone. The conic optimizer in MOSEK supports three structurally different types of cones K, which allows a surprisingly large number of nonlinear relations to be modeled (as described in the MOSEK modeling manual), while preserving the nice algorithmic and theoretical properties of linear optimization.

Fusion is a tool for building linear and conic optimization models in a simple and expressive manner using mainstream programming languages (Java, MATLAB, Microsoft .NET and Python). A typical (low-level) solver API requires the problem to be serialized into a single matrix and a few vectors, and constructing (or modifying) such a problem often proves to be both a time-consuming and error-prone process. Fusion, on the other hand, introduces a higher level of abstraction, which allows the user to focus explicitly on modeling oriented aspects rather than reformulating a given model for a particular solver API. For example, in Fusion it is easy to add variables and constraints to an existing model.

Typically a conic optimization model in Fusion can be developed in a fraction of the time compared to using a low-level C API, but of course Fusion introduces a computational overhead compared to customized C code. In most cases, however, the overhead is small compared to the overall solution time, and we generally recommend that Fusion is used as a first step to build and verify new models. Often, the final Fusion implementation will be directly suited for production code, and otherwise it readily provides a reference implemention for model verification.

The Fusion API is available for

- Java (1.6 and later),

- Microsoft .NET (2.1 and later)

- Python (2.5 and later), and

- MATLAB

These APIs are designed to be nearly identical, so an implementation can be easily ported between the supported platforms.

This manual assumes basic knowledge of Java development. When compiling examples, the relevant tools must be available, and a working MOSEK installation must be present (see "*The MOSEK installation manual*" included in the distro).

The Fusion/Java manual is included as PDF in the distribution, and an HTML version is available online at http://docs.mosek.com.

Code examples for Fusion/Java are located in

```
mosek/7/tools/examples/fusion/java
```

The examples directories contain scripts for building and running the various examples; these are called `build.bat` and `run.bat` on Windows, and `build.sh` and `run.sh` on Linux, Solaris and Mac OS X.

To run the Fusion/Java examples Java 1.5+ is required.

2.1   Limitations

Fusion imposes some limitations on certain aspects of a model:

- Constraints and variables belong to a single model, and cannot as such be used (e.g. stacked) with objects from other models.

- Constraint and variable domains are immutable. This means that it is not possible to change the bound type or value.

Furthermore, some features are on the drawing board but have not yet been implemented:

- Modification of the expression in constraints. Note that the objective can already be changed.

- Deletion of constraints. Note that deletion of variable is *not* planned.

- Additional methods for construction expressions.

2.2 Getting support

For information about obtaining technical support in relation to MOSEK products then please consult mosek.com/support/.

2.3 Getting started

We start our tutorial with two simple, yet representative Fusion examples. We do not discuss the implementation details of these examples at this point; rather they are included as a precursor to the more systematic discussion of the Fusion API ahead.

Before the examples can run it is necessary to have MOSEK and Python installed and set up. See "The MOSEK Installation manual" for details.

2.3.1 A conic quadratic example

A basic Markowitz portfolio optimization problem can be written (in conic form) as

$$
\begin{aligned}
\text{minimize} \quad & x^T \Sigma x \\
\text{subject to} \quad & \mu^T x \geq \delta \\
& e^T x = 1 \\
& x \geq 0,
\end{aligned}
$$

where $\Sigma = V^T V$ is a positive semidefinite covariance matrix. An equivalent conic formulation suited for Fusion is then

$$
\begin{aligned}
\text{minimize} \quad & t \\
\text{subject to} \quad & (1/2, t, Vx) \in \mathcal{Q}_r^n \\
& \mu^T x \geq \delta \\
& e^T x = 1 \\
& x \geq 0,
\end{aligned}
$$

where

$$\mathcal{Q}_r^n := \left\{ x \in \mathbb{R}^n \mid 2x_1 x_2 \geq \sum_{i=3}^{n} x_i^2, \ x_1, x_2 \geq 0 \right\}$$

is a *rotated quadratic cone.*

A Fusion implementation for this problem is given below.

```java
import mosek.fusion.*;

class markowitz1
{
  public static void main(String[] args)
    throws SolutionError
  {
    // Create a model object
    Model M = new Model("simple");
    try
    {
      // Add two variables
      Variable x = M.variable("x",1,Domain.inRange(0.0,1.0));
      Variable y = M.variable("y",1,Domain.unbounded());

      // Add a constraint on the variables
      M.constraint("bound y", Expr.sub(y,x), Domain.inRange(-1.0, 2.0));

      // Define the objective
      M.objective("obj", ObjectiveSense.Maximize, Expr.add(x,y));

      // Solve the problem
      M.solve();
    }
    finally
    {
      M.dispose();
    }
  }
}
```

The example can also be found in the distribution as `markowitz1` under

```
tools/examples/fusion/java
```

2.3.2   A semidefinite example

Suppose we are given a symmetric matrix $A \in \mathbb{R}^{n \times n}$, $A = A^T$. We may then be interested in computing the correlation matrix, which is nearest to A. This can be formulated as a semidefinite optimization problem

$$\begin{array}{ll} \text{minimize} & \|A - X\|_F \\ \text{subject to} & \mathbf{diag}(X) = e \\ & X \succeq 0. \end{array}$$

Let $\mathbf{vec}(\cdot)$ be a mapping from a symmetric matrix to a vector defined as

$$\mathbf{vec}(X) := (X_{11}, \sqrt{2}X_{21}, \ldots, \sqrt{X}_{n1}, X_{22}, \sqrt{2}X_{32}, \ldots, X_{nn}).$$

An equivalent conic formulation (exploiting symmetry of X) suited for Fusion is then

$$
\begin{aligned}
\text{minimize} \quad & t \\
\text{subject to} \quad & (t, z) \in \mathcal{Q}^n \\
& \mathbf{vec}(A - X) = z \\
& \mathbf{diag}(X) = e \\
& X \succeq 0.
\end{aligned}
$$

where

$$\mathcal{Q}^n := \left\{ x \in \mathbb{R}^n \mid x_1 \geq \sqrt{\sum_{i=2}^{n} x_i^2}, \ x_1 \geq 0 \right\}$$

is a *quadratic cone*.

A Fusion implementation for this problem is given below (for a specific choice of A).

```java
import mosek.fusion.*;

public class nearestcorr
{
  /** Assuming that e is an NxN expression, return the lower triangular part as a vector.
  */
  public static Expression vec(Expression e)
  {
    int N     = e.getShape().dim(0);
    int[] msubi = new int[N*(N+1)/2],
          msubj = new int[N*(N+1)/2];
    double[] mcof = new double[N*(N+1)/2];

    {
      for (int i = 0, k = 0; i < N; ++i)
        for (int j = 0; j < i+1; ++j, ++k)
        {
          msubi[k] = k;
          msubj[k] = i*N+j;
          if (i == j) mcof[k] = 1.0;
          else        mcof[k] = Math.sqrt(2);

        }
    }

    Matrix S = Matrix.sparse(N * (N+1) / 2,N * N,msubi,msubj,mcof);

    return Expr.mul(S,Expr.reshape( e, N * N ));
  }
  private static String arrtostr(double[] a)
  {
    StringBuilder b = new StringBuilder("[");
    java.util.Formatter f = new java.util.Formatter(b, java.util.Locale.US);
    if (a.length > 0) f.format(", %.3e", a[0]);
    for (int i = 1; i < a.length; ++i) f.format(", %.3e", a[i]);
```

```java
      b.append("]");
      return b.toString();
  }
  public static void main(String[] argv)
      throws SolutionError
  {
    int N = 5;
    double[][] A = new double[][]
        { { 0.0,  0.5,  -0.1,  -0.2,    0.5},
          { 0.5,  1.25, -0.05, -0.1,    0.25},
          {-0.1, -0.05,  0.51,  0.02, -0.05},
          {-0.2, -0.1,   0.02,  0.54, -0.1},
          { 0.5,  0.25, -0.05, -0.1,    1.25} };

    // Create a model with the name 'NearestCorrelation
    Model M = new Model("NearestCorrelation");
    try
    {
      // Setting up the variables
      Variable X = M.variable("X", Domain.inPSDCone(N));
      Variable t = M.variable("t", 1, Domain.unbounded());

      // (t, vec (A-X)) \in Q
      M.constraint( Expr.vstack(t, vec(Expr.sub(new DenseMatrix(A),X))), Domain.inQCone() );

      // diag(X) = e
      M.constraint(X.diag(), Domain.equalsTo(1.0));

      // Objective: Minimize t
      M.objective(ObjectiveSense.Minimize, t);

      // Solve the problem
      M.solve();

      // Get the solution values
      System.out.println("X = " + arrtostr(X.level()));

      System.out.println("t = " + arrtostr(t.level()));
    }
    finally
    {
      M.dispose();
    }
  }
}
```

The example can also be found in the distribution as `nearestcorr` under

`tools/examples/fusion/java`

2.3.3 Memory issues

Each `Model` object links to an external resource (the MOSEK task). Due to the way the garbage collector works it is not guaranteed when the `Model` object can be reclaimed, and in some cases it cannot be automatically reclaimed at all. This means that substantial amounts of memory may be leaked.

For this reason it is very important to always make sure that the Model object is properly disposed of when it is not used anymore by calling `Model.dispose()`.

The simplest way to ensure that `Model.dispose()` is called is to use the `using`-construction:

```
Model M = new Model();
try
{
  // do something with M
}
finally
{
  M.dispose();
}
```

This way `dispose()` will be called when the `try`-scope exits, even if an exception is thrown. If this cannot be used, e.g. if the Model object is returned by a factory function, the method must be explicitly called.

Furthermore, if the Model class is inherited and any work is done in the constructor, it is necessary to ensure that if an exception is thrown during construction, the `Model.dispose()` is called. To ensure this, use a construction along the lines

```
class MyModel extends Model
{
  public MyModel()
  {
    super();
    boolean finished = false;
    try
    {
      # perform initialization here
      finished = true;
    }
    finally
    {
      if (! finished)
        dispose()
    }
  }
}
...
{
  MyModel M = new MyModel();
  try
  {
    # do something with M
  }
  finally
  {
    M.dispose();
  }
}
```

Note that this construction aboids having to catch and rethrow an exception in the constructor, thus changing the exception stack trace.

2.4   Compiling and running examples

The examples directory includes scripts for compiling and running the examples on Windows (`build.bat` and `run.bat`) and Unix (`build.sh` and `run.sh`).

The build script compiles all exmaples into a single `.jar` file. To run the build script do:

- On Windows: Open a DOS box, go to the Fusion/Java examples directory and type

  ```
  build.bat
  ```

- On Linux, Mac OS X and Solaris: Open a terminal, go to the Fusion/Java examples directory and type

  ```
  bash build.sh
  ```

Once this has been done, the examples can be run. For example to run the example "diet":

- On Windows type

  ```
  run.bat diet
  ```

- On Linux, Mac OS X and Solaris type

  ```
  bash run.sh diet
  ```

The scripts contain the lines necessary to compile and run the examples, but the next sections show how to this for a single example `lo1_fusion.java` listed in the previous section.

2.4.1   Compiling and running on Windows

This requires that MOSEK is installed and works, and that a Oracle Java 1.5 or later is available.

To compile, open a DOS prompt (Select the `Start` menu, choose `run` and enter "`cmd`"), and change directory to where the Fusion/Java examples are located in the MOSEK installation, e.g., by typing

```
cd "C:\Program Files\mosek\7\tools\examples\fusion\java"
```

The example can then be compiled by typing

```
javac -classpath ..\..\..\platform\win64x86\bin\mosek.jar -d . lo1_fusion.java
```

This will create a file `com\mosek\fusion\examples\lo_fusion.class` under current directory. To run this example type

```
java -classpath ..\..\..\platform\win64x86\bin\mosek.jar;. com.mosek.fusion.examples.lo1_fusion
```

This will optimize the example and print the solution.

For 32 bit Windows replace `win64x86` with `win32x86`.

2.4.2   Compiling and running on Linux/Unix

This requires that MOSEK is installed and works, and that a Oracle Java 1.5 or later is available.

To compile, open a terminal and change directory to where the Fusion/Java examples are located in the MOSEK installation, e.g.,

```
cd $HOME/mosek/7/tools/examples/fusion/java
```

The example can now be compiled by typing

```
javac -classpath ../../../platform/linux64x86/bin/mosek.jar -d . lo1_fusion.java
```

This will create a file `com/mosek/fusion/examples/lo_fusion.class` under current directory. To run this example type

```
java -classpath ../../../platform/linux64x86/bin/mosek.jar:. com.mosek.fusion.examples.lo1_fusion
```

This will optimize the example and print the solution.

For other platforms than 64 bit linux, replace `linux64x86` with

- `linux32x86` for 32bit Linux,

- `osx32x86` for 32 bit Mac OS X,

- `osx64x86` for 64 bit Mac OS X,

- `solaris32x86` for 32 bit Solaris, and

- `solaris64x86` for 64 bit Solaris.

Chapter 3

Fusion tutorial

MOSEK/Fusion is an API for MOSEK that makes it easier to develop optimization models. Fusion is not a modeling language; rather it is an object oriented API which provides many of the same abstractions as modeling languages do. The fundamental construction in Fusion is a constraint that limits a linear expression to some domain:

$$Ax + b \in K, \ A \in \mathbb{R}^{m \times n}, \ b \in \mathbb{R}^m$$

where K is a convex set. Fusion supports only a small set of these sets directly, specifically, linearly bounded regions, quadratic cones and semidefinite cones, but these can be used to construct more complicated sets.

Furthermore, variables, expressions etc. in Fusion all have a *shape*, e.g., while the expression $Ax + b$ above is a one-dimensional vector, Fusion can manipulate variables, constraints and expressions of any number of dimensions. One example of this is the semidefinite variable, which is an $n \times n$ matrix-variable. Understanding how shapes work is important since they influence how expressions are built and used.

Fusion provides a library of functions to create complex expressions, and a low-level expression interface so other specialized constructions can be added.

3.1   Basic functionality

The API can be used directly to develop, manipulate and solve simple models, and for more complex tasks it is possible to extend the basic modeling objects to create functionality more suited for a specific problem.

The basic modeling objects in the API are

- `Model`. The model object containing all data, variables, constraints, the objective, solutions, etc.
- `Variable`. The object used as problem variables. Variables are n-dimensional objects and have a fixed domain.

Each variable belongs to exactly one model, so when a solution is available, it can be accessed through the variable object.

- `Constraint`. The object that results from creating a constraint in the model. It can be used to modify the constaint and access solution values.

  As with variables, each constraint belongs to exactly one model.

- `Domain`. This is the object that defines the various types of domains supported. The class contains various static methods acting as constructors.

- `Expression`. This is the abstract base class for all linear expression objects used in constraints. The actual expression objects are constructed by various static methods in the `Expr` class.

3.1.1   Variable objects

When a variable is created using `Model.variable` it will appear to simply be a `Variable` object. However, in reality all variables returned by `Model.variable` are `ModelVariable` objects; more specifically one of the descendants of `ModelVariable`. Furthermore, it is possible to create variable "views" by taking a slice (see `Variable.slice` and `SliceVariable`) of another variable or stacking (see `Variable.hstack`, `Variable.vstack` and `CompoundVariable`) several variable into a single variable vector.

The `Variable` class exposes the functionality common to all variable objects; primal and dual solution values, and the functionality for stacking and slicing. The specialized variable types derived from `Variable` provides access to more properties; bound values, and other solution values.

To access the properties particular to the actual variable type, it is necessary to up-cast the object to the relevant class.

All `ModelVariable`s are created with a shape, a domain and, optionally, a unique name, e.g.:

```
Variable y = M.variable("y", 5, Domain.greaterThan(0.0));
```

In this case

- `"y"` is the variable name, which must be unique among all variables in the model,

- `5` means that the variable is a vector containing 5 elements, and

- `Domain.greaterThan(0.0)` defines a linear domain of positive values.

The variable would implement the definition

$$y \in \mathbb{R}^5, \ y_i \geq 0, \ i = 1, \dots, 5$$

Variables with different shapes can be constructed by using a `Set` instead if an integer, e.g. a 2×2 matrix variable might be created like this:

```
M.variable("y2", new NDSet(2,2), Domain.greaterThan(0.0));
```

The shape serves two purposes:

- Together with the variable name the shape is used to create the names in the underlying model.

- They define the dimensions of the variable object; this influence how the variable is used when it is used as operand for multiplication, addition, subtraction, etc..

When the `Model` has been solved, the `Variable` object provides access to the solution values through `Variable.level` and `Variable.dual`.

3.1.1.1 Variable domains

When a variable is created, it is given a domain that defines which values are valid for the variable. For example, as we saw above, we can create a scalar variable $x \geq 0$ as

```
Variable x = M.variable("x", 1, Domain.greaterThan(0.0));
```

By specifying a size larger than 1, we can create a vector of these variables, and by replacing the size by a set, e.g.

```
M.variable("y2", new NDSet(2,2), Domain.greaterThan(0.0));
```

we can create a multidimensional variable, where each variable belongs to the given domain. The simple domains supported by Fusion are

- a fixed value $z = 0$, e.g.

  ```
  Variable z2 = M.variable(2, Domain.equalsTo(0.0));
  ```

- a vector of ranges, i.e., $b'_l \leq z \leq b'_u$,

  ```
  Variable z3 = M.variable(2, Domain.inRange(0.0, 10.0));
  ```

- a lower bound or an upper bound, i.e., $b'_l \leq z$

  ```
  Variable z5 = M.variable(2, Domain.greaterThan(0.0));
  ```

  or $z \leq b'_u$,

  ```
  Variable z1 = M.variable(2, Domain.lessThan(0.0));
  ```

The size and shape of these domains can be anything; vectors of arbitrary lenght, matrixes or multidimensional arrays. The bound values in the domains above can be

- a single value, which means that all elements have the same bound,

- a vector of values, the size of which must match the size of the variable, or

- a `Matrix` object, the dimensions of which must match exactly the dimensions of the variable.

Apart from these, Fusion supports three more complex domains:

- quadratic cones

$$
\mathcal{Q}^n := \left\{ (t, x) \in \mathbb{R} \times \mathbb{R}^n \mid t \geq 0,\, t \geq \sqrt{\sum_{i=1}^{n} x_i^2} \right\},
$$

where $n \geq 1$, e.g.,

```
Variable zc1 = M.variable("zc1", 3, Domain.inQCone());
```

- rotated quadratic cones

$$\mathcal{Q}_r^n := \left\{ (t,x) \in \mathbb{R}^2 \times \mathbb{R}^n \mid t_1, t_2 \geq 0,\ 2t_1 t_2 \geq \sum_{i=1}^{n} x_i^2 \right\},$$

where $n \geq 2$, e.g.,

```
Variable zc2 = M.variable("zc2", 3, Domain.inRotatedQCone());
```

- cones of symmetric positive definite matrices

$$\mathcal{S}_+^n := \left\{ X \in \mathbb{R}^{n \times n} \mid X^T = X,\ y^T X y \geq 0, \forall y \in \mathbb{R}^n \right\}$$

where $n \geq 1$, e.g.,

```
Variable zs  = M.variable("zs", new NDSet(3,3), Domain.inPSDCone());
```

The variables in the two quadratic cones can either be vectors or matrixes. This is interpreted as follows: If $x \in \mathbb{R}^n$, then the x domains `Domain.inQCone` and `Domain.inRotatedQCone` are interpreted as the constraints

$$x_1 \geq \sqrt{\sum_{j=2}^{n} x_j^2}$$

and

$$2x_1 x_2 \geq \sum_{j=3}^{n} x_j^2,$$

respectively.

If the variable is a matrix, i.e. $x \in \mathbb{R}^{m \times n}$, then the conic quadratic domains are interpreted as

$$x_{i1} \geq \sqrt{\sum_{j=2}^{n} x_{ij}^2},\ i = 1, \ldots, m$$

and

$$2x_{i1} x_{i2} \geq \sum_{j=3}^{n} x_{ij}^2,\ i = 1, \ldots, m.$$

Similarly, for the semidefinite cone, if $x \in \mathbb{R}^{n \times n}$, then the x domain `Domain.inPSDCone` is interpreted as
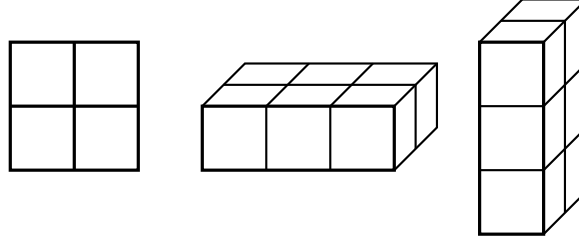
Figure 3.1: Examples of variable or constraint shapes.

$$x \in \mathcal{S}_+^n,$$

while if $x \in \mathbb{R}^{m \times n \times n}$, then the domain is interpreted as

$$x_i \in \mathcal{S}_+^n, \; i = 1, \ldots, m.$$

3.1.1.2 Variable shapes, slicing and stacking

Since Fusion deals with variables on a block-level, we need to keep in mind their shapes.

Variables are shaped like n-dimensional boxes with a finite number of elements in each dimension. See Fig. 3.1 for an example. This illustrates: 1) A 2×2 shape, 2) a 1×3×2 shape, and 3) a 3×1×2 shape and would be written as

```
M.variable(new NDSet(2,2), Domain.unbounded());
M.variable(new NDSet(1,3,2), Domain.unbounded());
M.variable(new NDSet(3,1,2), Domain.unbounded());
```

Variables can be sliced. A slice variables represents a sub-block of a variable block (see Fig. 3.2), which would be written as

```
v = M.variable(new NDSet(4,4),Domain.unbounded());
v.slice(new int[] {0,0}, new int[] {2,2});
v.index(4,4);
```

The slice has the same number of dimensions as the original object.

Variables can be stacked to form new variables. When stacking variables or using them in expressions it is necessary that the shapes match for the operation. There are various methods for stacking variables, specifically `Variable.hstack`, `Variable.vstack` and `Variable.stack`:

- Stacking two one-dimensional variables with `Variable.vstack` (see Fig. 3.3) written as

  ```
  Variable a = M.variable(3,Domain.unbounded());
  Variable b = M.variable(3,Domain.unbounded());
  Variable.vstack(a,b);
  ```

- Stacking two three-dimensional variables with `Variable.stack` where their shapes differ only in the first dimension (see Fig. 3.4) written as
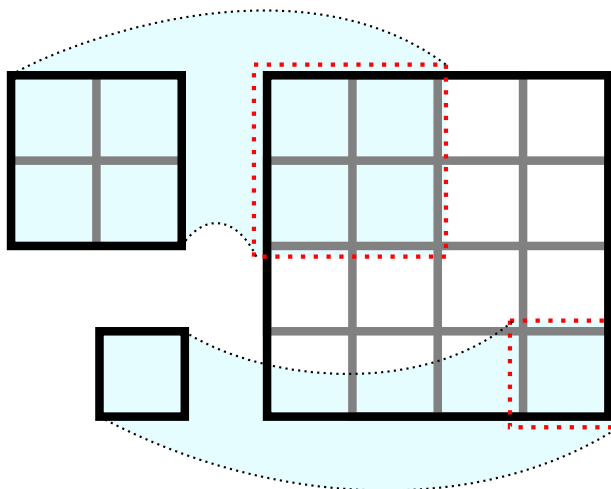
Figure 3.2: How shaped elements are mapped to linear indexes.
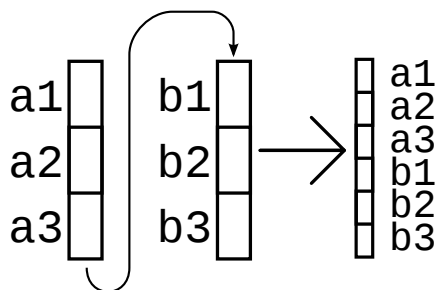
Figure 3.3: Stacking one-dimensional objects.
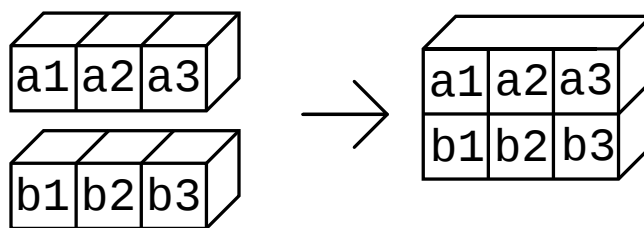
Figure 3.4: Stacking three-dimensional objects.
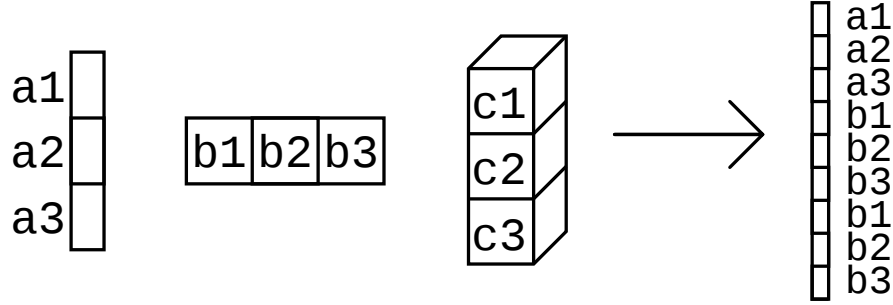
Figure 3.5: Stacking three vectors.
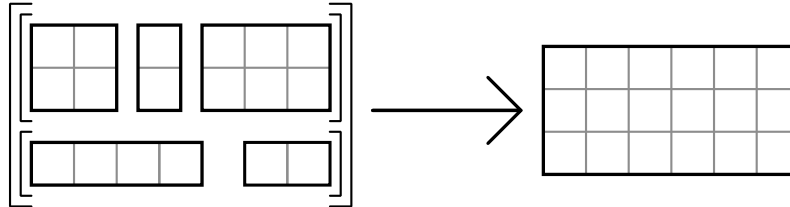
Figure 3.6: Stacking in two dimensions.

```
Variable a = M.variable(new NDSet(1,3),Domain.unbounded());
Variable b = M.variable(new NDSet(1,3),Domain.unbounded());
Variable.vstack(a,b);
```

- Stacking three variables with `Variable.vstack`, where at most one dimension is greater than one (see Fig. 3.5) written as

```
Variable a = M.variable(3,Domain.unbounded());
Variable b = M.variable(new NDSet(1,3),Domain.unbounded());
Variable c = M.variable(3,Domain.unbounded());
Variable.vstack(a,b,c);
```

- Stacking variables in two dimensions using `Variable.stack`. In this case the height (second dimension) of variables in each row must match, and the total length of each row must match (see Fig. 3.6) written as

```
Variable a1 = M.variable(new NDSet(2,2),Domain.unbounded());
Variable a2 = M.variable(new NDSet(2,1),Domain.unbounded());
Variable a3 = M.variable(new NDSet(2,3),Domain.unbounded());
Variable b1 = M.variable(new NDSet(1,4),Domain.unbounded());
Variable b2 = M.variable(new NDSet(1,2),Domain.unbounded());
Variable.stack(new Variable[][] { { a1, a2, a3 }, { b1, b2 }});
```

3.1.2   Creating expressions and constraints

Creating a constraint or an objective function requires an *expression*. The Expression object represents an array of scalar linear expressions

$$a^T x + b, \ a \in \mathbb{R}^n, \ x \in \mathbb{R}^n, \ b \in \mathbb{R}.$$

Expressions can be shaped the same way as variables; as scalars, vectors, matrixes or multi-dimensional arrays.

In the simplest case the expression is a dense vector, but it may also have a shape and a sparsity pattern. Consider the following case:

```
Matrix m = new DenseMatrix(new double[][] { {1.1,1.2},{2.1,2.2}});
Variable x = M.variable(2,Domain.unbounded());
Variable v = M.variable(2, Domain.greaterThan(0.0));
Variable w = M.variable(new NDSet(2,2), Domain.greaterThan(0.0));
Expression e1 = Expr.mul(m,v);
Expression e2 = Expr.mul(m,w);
```

Here we have

$$\texttt{e1} = \left[\begin{array}{cc} m_{11} & m_{12} \\ m_{21} & m_{22} \end{array} \right] \left[\begin{array}{c} v_1 \\ v_2 \end{array} \right]$$

and

$$\texttt{e2} = \left[\begin{array}{cc} m_{11} & m_{12} \\ m_{21} & m_{22} \end{array} \right] \left[\begin{array}{cc} w_{11} & w_{12} \\ w_{21} & w_{22} \end{array} \right]$$

so

$$\texttt{e1} \in \mathbb{R}^2 \text{ and } \texttt{e2} \in \mathbb{R}^{2 \times 2}$$

Since m is dense, both e1 and e2 are dense, i.e., all elements are non-zero. If the matrix m is sparse, the resulting Expression will be sparse. If these expressions are used when creating a constraint, e.g., as

```
M.constraint(e2, Domain.equalsTo(5.0));
```

the resulting Constraint will get the same shape as e2.

The Expr class can be explicitly instantiated from array data (see Expr.Expr), but also contains static methods for building expressions, e.g., the basic operations:

Expr.add(lhs,rhs)

> Add two expressions, or add a constant or a constant vector to an expression.

Expr.sub(lhs,rhs)

> Subtract two expressions, or subtract a constant or a constant vector from an expression.

Expr.mul(lhs,rhs)

> Multiply a scalar, a vector or a matrix by a variable.

`Expr.dot`(lhs,rhs)

> Dot product of two items. The result is an expression of size 1.

`Expr.sum`(rhs)

> Sum of the elements in an expression vector.

A *constraint* is an expression with a bound or domain, generally put,

$$Ax + b \in K$$

where K is one of a few sets supported by Fusion, specifically,

- a point, yielding a constraint of the form $Ax + b = b'$,

- a vector of ranges, i.e., $b'_l \le Ax + b \le b'_u$,

- a lower bound or an upper bound, i.e., $b'_l \le Ax + b$ or $Ax + b \le b'_u$,

- a quadratic cone or a rotated quadratic cone,

- a semidefinite cone,

or it may be unbounded.

The size and shape of the expression and the domain must match; for example, if the domain is a 4-dimensional quadratic cone, the expression must be a vector of length 4. In most cases it is not necessary to write the size of the domain explicitly when it can be deduced from the expression. Suppose that we wish to write

$$Ax = \left[\begin{array}{c} 2 \\ 2 \end{array} \right]$$

where Ax is a vector of size 2. Then we could write

```
M.constraint(Expr.mul(A,x), Domain.equalsTo(new double[] { 2.0, 2.0 }));
```

but Fusion also allows the more convenient form

```
M.constraint(Expr.mul(A,x), Domain.equalsTo(2.0));
```

This is discussed in more detail in 3.2.

Constraints are added to a model using the `Model.constraint` method, which creates a new `Constraint` object.

3.1.2.1 The constraint object

The Fusion constraint mirrors the behaviour of the variable. Constraints are created using `Model.constraint`, which returns a `Constraint` object. The real type of this object is a type derived from `ModelConstraint`, specifically one of the types

- `LinearConstraint`

- `ConicConstraint`

- `RangedConstraint`

- `PSDConstraint`

depending on the domain. As for variables, these can be sliced (using `Constraint.slice`) and stacked (using `Constraint.stack`), which will create objects of the types

- `SliceConstraint`

- `CompoundConstraint`.

A `Constraint` has two solution values. The primal, accessed though `Constraint.level`, corresponds to the value of the constraint expression at the solution, and the dual, accessed through `Constraint.dual`.

3.1.3   Solving and using solutions

When a `Model` has been solved, the solution values can be accessed through the `Variable` and `Constraint` objects. The concept of *solution* in Fusion mirrors that of MOSEK, i.e., a *solution* may in fact be either a set of values defining the optimal solution, or a set of values that proves that the problem is infeasible (a certificate of infeasibility).

By default, asking for a non-optimal solution will cause an exception.

3.1.3.1   Accessing solutions

Each variable and constraint has two solution sets; a primal and a dual. These can be accessed through the two methods `level` and `dual`:

`Variable.level`

> Primal solution values of the variable.
>
> For optimal solutions this the value that solves the problem implemented by the `Model`.
>
> For a certificate of dual infeasibility the level value defines the certificate. For a certificate of dual infeasibility the level values are undefined.

`Variable.dual`

> Dual solution values of the variable.
>
> For optimal solutions this the value that solves the problem implemented by the `Model`.
>
> For a certificate of primal infeasibility the dual values of all variables together with the dual values of all constraints define the certificate. For a certificate of dual infeasibility the dual values are undefined.

Constraint.level

> Primal solution values of the constraint. For optimal solutions this corresponds to the expression of the constraint evaluated at the solution point.

Constraint.dual

> Dual solution values of the constraint.
>
> The dual values work as the dual values of variables.

Solutions are returned as an array, even for variables or constraints that are multi-dimensional, and all values are returned, even if only a few elements have non-zero values. For example, for a 3×3 variable where only element $(2,2)$ is used in the problem, the **level** method would return an array of length 9. The order of the elements in this array would be

$$[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]$$

In these cases it may be easier to create slices of the variable or constraint using **Variable.slice** or **Constraint.slice** and access the solution through them.

3.1.4 Example: Baker

The following is a very small example of two variables and three constraints.

The problem is this: A baker has 150 kg flour, 22 kg sugar and 25 kg butter in stock. Furthermore, he has two recipes: *Cakes* made from 3 kg of flour, 1.0 kg of sugar and 1.2 kg of butter per dozen, and *breads* made from 5.0 kg of flour, 0.5 kg of sugar and 0.5 kg of butter per dozen. The revenue is \$4 per dozen cakes and \$6 per dozen breads. What combination of cakes and breads will then maximize the baker's revenue?

The linear program for this problem can be formulated as follows:

$$\begin{aligned}
\text{minimize} \quad & 4x_1 + 6x_2 \\
\text{subject to} \quad & 3x_1 + 5x_2 \leq 150 \\
& 1x_1 + 0.5x_2 \leq 22 \\
& 1.2x_1 + 0.5x_2 \leq 25 \\
& x_1, x_2 \geq 0
\end{aligned}$$

Here, x_1, x_2 denotes the number of cakes and breads respectively.

In Fusion all variables, constraints, expressions, etc., are in fact vectors or matrices; for example we might add a variable **production** of size 2 rather than adding two separate variables. In our implementation we will first define data separately:

```
private static String[] ingredientnames = { "Flour", "Sugar", "Butter" };
private static double[] stock = { 150.0,   22.0,   25.0 };

private static double[][] recipe_data =
  { { 3.0, 5.0 },
    { 1.0, 0.5 },
    { 1.2, 0.5 } };
```

```java
private static String[] productnames = { "Cakes","Breads" };

private static double[] revenue = { 4.0, 6.0 };
```

The variable `production` is defined as a vector indexed over the available products:

```java
Variable production = M.variable("production",
                                 new StringSet(productnames),
                                 Domain.greaterThan(0.0));
```

With this we can define the objective and constraints as:

```java
M.objective("revenue",
            ObjectiveSense.Maximize,
            Expr.dot(revenue, production));
```

and

```java
M.constraint(Expr.mul(recipe, production), Domain.lessThan(stock));
```

3.1.4.1   source code: Baker

```java
//
// Copyright: Copyright (c) MOSEK ApS, Denmark. All rights reserved.
//
// File:      baker.java
//
// Purpose: Demonstrates a small linear problem.
//
// Source: "Linaer Algebra" by Knut Sydsaeter and Bernt Oeksendal.
//
// The problem: A baker has 150 kg flour, 22 kg sugar, 25 kg butter and two
// recipes:
//    1) Cakes, requiring 3.0 kg flour, 1.0 kg sugar and 1.2 kg butter per dozen.
//    2) Breads, requiring 5.0 kg flour, 0.5 kg sugar and 0.5 kg butter per dozen.
// Let the revenue per dozen cakes be $4 and the revenue per dozen breads be $6.
//
// We now wish to compute the combination of cakes and breads that will optimize
// the total revenue.


package com.mosek.fusion.examples;
import mosek.fusion.*;
import java.io.PrintWriter;

public class baker
{
    private static String[] ingredientnames = { "Flour", "Sugar", "Butter" };
    private static double[] stock = { 150.0,    22.0,     25.0 };

    private static double[][] recipe_data =
      { { 3.0, 5.0 },
        { 1.0, 0.5 },
        { 1.2, 0.5 } };
    private static String[] productnames = { "Cakes","Breads" };

    private static double[] revenue = { 4.0, 6.0 };
    public static void main(String[] args)
```

```java
    throws SolutionError
{
    Matrix recipe = new DenseMatrix(recipe_data);
    Model M = new Model("Recipe");
    try
    {
// "production" defines the amount of each product to bake.
    Variable production = M.variable("production",
                                     new StringSet(productnames),
                                     Domain.greaterThan(0.0));
    // The objective is to maximize the total revenue.
    M.objective("revenue",
                ObjectiveSense.Maximize,
                Expr.dot(revenue, production));

    // The prodoction is constrained by stock:
    M.constraint(Expr.mul(recipe, production), Domain.lessThan(stock));
    M.setLogHandler(new PrintWriter(System.out));

    // We solve and fetch the solution:
    M.solve();
    double[] res = production.level();
    System.out.println("Solution:");
    for (int i = 0; i < res.length; ++i)
    {
        System.out.println(" Number of " + productnames[i] + " : " + res[i]);
    }
    System.out.println(" Revenue : $" +
                    (res[0] * revenue[0] + res[1] * revenue[1]));
    }
    finally
    {
      M.dispose();
    }
  }
}
```

3.1.5   Compiling and running the example

The Fusion API is included in the `mosek.jar` located in the `bin/` directory in the distro.

3.1.5.1   MS Windows

To build the example on Windows, open a terminal and go to the Fusion/Java examples directory. Then type

```
 javac -classpath "C:\Program Files\mosek\7\tools\platform\win64x86\bin\mosek.jar" -d . baker.java
```

Then, to run it by typing

```
 java -classpath "C:\Program Files\mosek\7\tools\platform\win64x86\bin\mosek.jar;." com.mosek.fusion.examples.baker
```

3.1.5.2   Linux and Mac OS X

Open a terminal and go to the Fusion/Java examples directory. Then type

```
javac -classpath "$HOME/mosek/7/platform/win64x86/bin/mosek.jar" -d . baker.java
```

Then, to run it by typing

```
java -classpath "$HOME/mosek/7/platform/win64x86/bin/mosek.jar:." . com.mosek.fusion.examples.baker
```

3.2   Shapes and sparsity

The shapes of variables, expressions and constraints in Fusion work as scalar elements stacked in multiple dimensions. In one dimension they work as vectors, in two as matrixes.

Once created, the shape of an item is immutable. This may seem as a problem, e.g. when using column-generation methods or when building a model without initially knowing the ultimate number of variables, but Fusion only uses memory for the elements that are used in the model. For example, declaring a variable as

```
M.variable(new NDSet(10000, 100000), Domain.unbounded());
```

will only require a small constant amount of memory. This means that when a method requires addition of new elements to an existing variable, the solution is to simply initially declare the variable "large enough".

There are some hard limits on shapes and sizes in Fusion:

- The maximum number of variable elements used in a model can be no larger than $2^{31} - 1$.

- The maximum size of a dimension is $2^{31} - 1$.

- For efficiency reasons the total size of an item (the product of the dimensions) is limited to $2^{63} - 1$.

Fetching a solution from a shaped variable produces a flat array of values. This means that *all* values, even the ones that are not used in the problem, are returned, and that the variable elements are linearly indexed. In this case, it is better to create a slice variable holding the relevant elements and fetch the solution for this; fetching the full solution may well cause an exception due to memory exhaustion or platform-dependant constraints on array sizes.

3.3   Duality

Duality in Fusion mirrors duality concepts in standard conic optimization. For example, consider a linear optimization problem in standard form

$$
\begin{array}{ll}
\text{minimize} & c^T x \\
\text{subject to} & Ax = b \\
& x \geq 0,
\end{array}
$$

with a dual problem

$$\begin{aligned}
\text{maximize} \quad & b^T y \\
\text{subject to} \quad & c - A^T y = s \\
& s \geq 0.
\end{aligned}$$

We call s and y *dual variables* (or Lagrange multipliers) associated with the primal variable x and constraints $Ax = b$, respectively. A well-known fact about linear optimization is the *strong duality* property, which states that for an optimal solution we have $c^T x = b^T y$ (zero duality-gap) and $x^T s = 0$ (complementarity of x and s).

As an example we consider a simple optimization problem

$$\begin{aligned}
\text{minimize} \quad & x_1 + x_2 \\
\text{subject to} \quad & -(1/2)x_1 + x_2 = 1 \\
& x_1, x_2 \geq 0,
\end{aligned}$$

with a corresponding dual problem

$$\begin{aligned}
\text{maximize} \quad & y \\
\text{subject to} \quad & 1 + (1/2)y = s_1 \\
& 1 - y = s_2 \\
& s_1, s_2 \geq 0.
\end{aligned}$$

By inspection, we see that the optimal values to the primal and dual problems are $(x_1, x_2) = (0, 1)$ and $y = 1$, $(s_1, s_2) = (3/2, 0)$ respectively. We note that the duality gap is zero

$$1 \cdot x_1 + 1 \cdot x_2 = 1 \cdot y = 1$$

and that x and s are complementary,

$$x_1 \cdot s_1 + x_2 \cdot s_2 = 0 \cdot (3/2) + 1 \cdot 0 = 0.$$

3.3.1   Source code: duality example

```
package com.mosek.fusion.examples;

import mosek.fusion.*;

public class duality
{
    public static void main(String[] args)
      throws SolutionError
    {
      double[][] A =  { { -0.5, 1.0 }  };
      double[]   b =  { 1.0 };
      double[]   c =  { 1.0, 1.0 };

      Model M = new Model("duality");
```

```
    try
    {
      Variable x = M.variable("x", 2, Domain.greaterThan(0.0));

      Constraint con = M.constraint(Expr.sub(Expr.mul(new DenseMatrix(A), x), b), Domain.equalsTo(0.0));

      M.objective("obj", ObjectiveSense.Minimize, Expr.dot(c, x));

      M.solve();
      double[] xsol = x.level();
      double[] ssol = x.dual();
      double[] ysol = con.dual();

      System.out.printf("x1,x2,s1,s2,y = %e, %e, %e, %e, %e\n",xsol[0],xsol[1],ssol[0],ssol[1],ysol[0]);
    }
    finally
    {
      M.dispose();
    }
  }
}
```

3.4 Interacting with the solver

In a few cases it is necessary to interact directly with the MOSEK solver.

3.4.1 Enabling log output

The log output from the solver can be intercepted by setting a handler for the log stream. This is done
with the `Model.setLogHandler` function:

```
 M.setLogHandler(new PrintWriter(System.out));
```

The `logHandler` object is an `java.io.Writer` object.

3.4.2 Dump the optimization model to a file

The optimization problem can be written to a file. This is particularly useful when debugging a model
or exporting a problem to other solvers than MOSEK. This can be done using the `Model.writeTask`
method, that requires the user to provide the destination file name.

The file type is deduced from the file extension. The available file types are

Type	extension	Problem type
LP	.lp	(mixed-integer) linear,quadratic
MPS	.mps	(mixed-integer) linear,quadratic
OPF	.opf	(mixed-integer) linear,quadratic,conic quadratic
Task	.task	all available in MOSEK
OSiL	.xml	(mixed-integer) linear

For instance, to dump an optimization model `M` to an `OPF` file named `my-problem` we may write

```
M.writeTask('my-problem.opf');
```

The task file format includes also the solver parameters and this is particularly useful when debugging and posting a support request to MOSEK, because it helps exactly reproducing the solver behavior.

All file type can be write in `gzip` compressed format specifying the `.gz` extension. For instance the previous code becomes

```
M.writeTask('my-problem.opf.gz');
```

The `OPF` and `Task` formats also store the last optimal solution available form the given optimization problem.

3.4.3 Setting solver parameters

MOSEK solver parameters can be changed directly from Fusion using `Model.setSolverParam`. For example, we can set the mixed integer stopping criteria (solution quality and time limit) as

```
// Set max solution time
M.setSolverParam("mioMaxTime", 60.0);
// Set max relative gap (to its default value)
M.setSolverParam("mioTolRelGap", 1e-4);
// Set max absolute gap (to its default value)
M.setSolverParam("mioTolAbsGap", 0.0);
```

Parameters are set using a name and a value, which may be a string, an integer or a floating point value. Note that e.g. a parameter that accepts integer values can be set to a string value if the string can be converted to an integer.

See Sec. 5.42 for the complete list of all recognized parameters.

3.5 Integer optimization

Integer conic problems are specified similarly as conic problems with the exception that some variables are constrained to be integer. As an example, consider the integer linear optimization problem

$$
\begin{array}{ll}
\text{minimize} & x_1 + 0.64x_2 \\
\text{subject to} & 50x_1 + 31x_2 \le 250 \\
& 3x_1 - 2x_2 \ge -4 \\
& x_1, x_2 \ge 0, \\
& x_1, x_2 \text{ are integers.}
\end{array}
$$

The integer variables are specified as:

```
Variable x = M.variable("x", 2, Domain.greaterThan(0.0), Domain.isInteger());
```

Binary variables can be constructed in a similar way by additionally constraining them to be ranged variables

$$x_i \in [0,1], \ x_i \ \text{ is integer.}$$

Additionally, we set parameters specific for the MOSEK solver:

```java
// Set max solution time
M.setSolverParam("mioMaxTime", 60.0);
// Set max relative gap (to its default value)
M.setSolverParam("mioTolRelGap", 1e-4);
// Set max absolute gap (to its default value)
M.setSolverParam("mioTolAbsGap", 0.0);
```

Changing such parameters may be necessary to limit the solution time of difficult integer problems; we refer to the MOSEK reference manual for further details.

Finally, note that duality information is not available for integer optimization problems. However, some duality information can be obtained by fixing integer variables to their optimal values, and resolving the problem without integer constraints.

3.5.1   source code: milo1

```java
package com.mosek.fusion.examples;
//
//    Copyright: Copyright (c) MOSEK ApS, Denmark. All rights reserved.
//
//    File:     milo1_fusion.java
//
//    Purpose:  Demonstrates how to solve a small mixed
//              integer linear optimization problem.
//
import mosek.fusion.*;

public class milo1_fusion
{
    public static void main(String[] args)
      throws SolutionError
    {
        double[][] A =
            { { 50.0, 31.0 },
              { 3.0,  -2.0 } };
        double[] c = { 1.0, 0.64 };

        Model M = new Model("milo1");
        try
        {
          Variable x = M.variable("x", 2, Domain.greaterThan(0.0), Domain.isInteger());

          // Create the constraints
          //      50.0 x[0] + 31.0 x[1] <= 250.0
          //       3.0 x[0] -  2.0 x[1] >= -4.0
          M.constraint("c1", Expr.dot(A[0], x), Domain.lessThan(250.0));
          M.constraint("c2", Expr.dot(A[1], x), Domain.greaterThan(-4.0));

          // Set max solution time
          M.setSolverParam("mioMaxTime", 60.0);
          // Set max relative gap (to its default value)
          M.setSolverParam("mioTolRelGap", 1e-4);
          // Set max absolute gap (to its default value)
          M.setSolverParam("mioTolAbsGap", 0.0);
```

```java
          // Set the objective function to (c^T * x)
          M.objective("obj", ObjectiveSense.Maximize, Expr.dot(c, x));

          // Solve the problem
          M.solve();

          // Get the solution values
          double[] sol = x.level();
          System.out.printf("MIP rel gap = %.2f (%f)\n",M.getSolverDoubleInfo("mioObjRelGap"),M.getSolverDoubleInfo("mioObj
          System.out.printf("x1,x2 = %e, %e\n",sol[0],sol[1]);
        }
      finally
        {
          M.dispose();
        }
    }
}
```

Chapter 4

Case Studies

4.1 Stingler's Diet Model

Stingler's diet model describes how to construct a diet from various foods so that a certain nutrient intake is guaranteed. The model is based on numbers for a moderately active male weighting 70 kg, and the nutrient requirements are based on recommendations by the National Research Council in 1943. Prices are also based on 1943 prices.

We consider a set of foods and a *daily recommendation*, each with a set of nutrient values per $ spent:

Food - Nutrient	energy (kcal)	protein (g)	calcium (mg)	iron (g)	vitamin a (mg)	vitamin b1 (1000 IUs)	vitamin b2 (mg)	niacin (mg)	vitamin c (mg)
Daily rec.	3.0	70	.8	12	5.0	1.8	2.7	18	75
wheat	44.7	1411	2.0	365	0	55.4	33.3	441	0
cornmeal	36	897	1.7	99	30.9	17.4	7.9	106	0
cannedmilk	8.4	422	15.1	9	26	3	23.5	11	60
margarine	20.6	17	.6	6	55.8	.2	0	0	0
cheese	7.4	448	16.4	19	28.1	.8	10.3	4	0
peanut butter	15.7	661	1	48	0	9.6	8.1	471	0
lard	41.7	0	0	0	.2	0	.5	5	0
liver	2.2	333	.2	139	169.2	6.4	50.8	316	525
porkroast	4.4	249	.3	37	0	18.2	3.6	79	0
salmon	5.8	705	6.8	45	3.5	1	4.9	209	0
greenbeans	2.4	138	3.7	80	69	4.3	5.8	37	862
cabbage	2.6	125	4	36	7.2	9	4.5	26	5369
onions	5.8	166	3.8	59	16.6	4.7	5.9	21	1184
potatoes	14.3	336	1.8	118	6.7	29.4	7.1	198	2522
spinach	1.1	106	0.0	138	918.4	5.7	13.8	33	2755
sweet potatos	9.6	138	2.7	54	290.7	8.4	5.4	83	1912
peaches	8.5	87	1.7	173	86.8	1.2	4.3	55	57
prunes	12.8	99	2.5	154	85.7	3.9	4.3	65	257
limabeans	17.4	1055	3.7	459	5.1	26.9	38.2	93	0
navybeans	26.9	1691	11.4	792	0	38.4	24.6	217	0

We now wish to find the cheapest way to cover the daily recommendation.

Let f_{ij} denote the value of nutrient $j \in \mathcal{N}$ in food $i \in \mathcal{F}$, and let r_j be the daily recommended allowance if nutrient j. Then we can write the problem as:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i \in \mathcal{F}} x_i \\
\text{subject to} \quad & \sum_{i \in \mathcal{F}} x_i f_{ij} \geq r_j, \ j \in \mathcal{N} \\
& x_i \in \mathbb{R}_+, i \in \mathcal{F}
\end{aligned}
\tag{4.1}
$$

We implement this model slightly differently than the *Baker* example: We create a specialized `DietModel` class that extends the `Model` class, and can be instantiated with the problem data. This is a simple way to allow separation of the model and data, and encapsulation of the data checking.

The complete implementation is included below.

4.1.1   Source code: Diet Model

```
// Copyright: Copyright (c) MOSEK ApS, Denmark. All rights reserved.
//
// File:      diet.py
//
```

```java
// Purpose: Solving Stigler's Nutrition model (DIET,SEQ=7)
//
// Source: GAMS Model library,
//          Dantzig, G B, Chapter 27.1. In Linear Programming and Extensions.
//          Princeton University Press, Princeton, New Jersey, 1963.
//
// Given a set of nutrients, foods with associated nutrient values, allowance of
// nutrients per day, the model will find the cheapest combination of foods
// which will provide the necessary nutrients.
//
//  Arguments for construction:
//  name      - Model name.
//  foods     - List of M names of the foods.
//  nutrients - List of N names of the nutrients.
//  daily_allowance - List of N floats denoting the daily allowance of each
//              nutrient.
//  nutritive_value - Two-dimensional MxN array of floats where each row
//              denotes the nutrient values for a single food per $ spent.
package com.mosek.fusion.examples;
import mosek.fusion.*;
/*

NOT IMPLEMENTED YET

*/

public class diet extends Model
{
  public class DataException extends RuntimeException
  {
    public DataException(String msg) { super(msg); }
  }

  public double[] dailyAllowance;
  public Matrix nutrientValue;

  private Variable   dailyPurchase;
  private Constraint dailyNutrients;

  public diet(String     name,
              String[]   foods,
              String[]   nutrients,
              double[]   daily_allowance,
              double[][] nutritive_value)
  {
    super(name);
    boolean finished = false;
    try
    {
      dailyAllowance = daily_allowance;
      nutrientValue = (new DenseMatrix(nutritive_value)).transpose();

      int M = foods.length;
      int N = nutrients.length;

      if (dailyAllowance.length != N)
        throw new DataException(
          "Length of daily_allowance does not match the number of nutrients");
```

```java
      if (nutrientValue.numColumns() != M)
        throw new DataException(
          "Number of rows in nutrient_value does not match the number of foods");
      if (nutrientValue.numRows() != N)
        throw new DataException(
          "Number of columns in nutrient_value does not match the number of nutrients");

      dailyPurchase = variable("Daily Purchase",
                              new StringSet(foods),
                              Domain.greaterThan(0.0));
      dailyNutrients =
          constraint("Nutrient Balance",
                    new StringSet(nutrients),
                    Expr.mul(nutrientValue,dailyPurchase),
                    Domain.greaterThan(dailyAllowance));
      objective(ObjectiveSense.Minimize, Expr.sum(dailyPurchase));
      finished = true;
    }
    finally
    {
      if (! finished)
        dispose();
    }
  }

  public double[] dailyPurchase()
    throws SolutionError
  {
    return dailyPurchase.level();
  }

  public double[] dailyNutrients()
    throws SolutionError
  {
    return dailyNutrients.level();
  }


  /* Main class with data definitions */
  public static void main(String[] argv)
  {
    String[] nutrient_unit = {
        "thousands",  "grams",         "grams",
        "milligrams", "thousand ius", "milligrams",
        "milligrams", "milligrams",    "milligrams" };
    String[] nutrients = {
        "calorie",     "protein",       "calcium",
        "iron",        "vitamin a",     "vitamin b1",
        "vitamin b2", "niacin",         "vitamin c" };
    String[] foods = {
        "wheat",          "cornmeal", "cannedmilk", "margarine", "cheese",
        "peanut butter","lard",       "liver",      "porkroast", "salmon",
        "greenbeans",    "cabbage",   "onions",      "potatoes",  "spinach",
        "sweet potatos","peaches",   "prunes",       "limabeans", "navybeans" };
    double[][] nutritive_value = {
//  calorie       calcium      vitamin a        vitamin b2      vitamin c
//         protein        iron           vitamin b1       niacin
      {44.7,  1411,   2.0,   365,    0,      55.4,   33.3,  441,     0}, // wheat
```

```
        {36,      897,   1.7,    99,    30.9,    17.4,    7.9,   106,       0},  // cornmeal
        { 8.4,    422,  15.1,     9,    26,        3,    23.5,    11,      60},  // cannedmilk
        {20.6,     17,    .6,     6,    55.8,      .2,     0,      0,       0},  // margarine
        { 7.4,    448,  16.4,    19,    28.1,      .8,    10.3,    4,       0},  // cheese
        {15.7,    661,   1,      48,     0,       9.6,    8.1,   471,       0},  // peanut butter
        {41.7,      0,    0,      0,      .2,      0,       .5,    5,       0},  // lard
        { 2.2,    333,    .2,   139,   169.2,     6.4,    50.8,  316,     525},  // liver
        { 4.4,    249,    .3,    37,     0,      18.2,     3.6,   79,       0},  // porkroast
        { 5.8,    705,   6.8,    45,     3.5,      1,      4.9,   209,      0},  // salmon
        { 2.4,    138,   3.7,    80,    69,        4.3,    5.8,    37,     862},  // greenbeans
        { 2.6,    125,   4,      36,     7.2,      9,      4.5,    26,    5369},  // cabbage
        { 5.8,    166,   3.8,    59,    16.6,      4.7,    5.9,    21,    1184},  // onions
        {14.3,    336,   1.8,   118,     6.7,     29.4,    7.1,   198,    2522},  // potatoes
        { 1.1,    106,   0.0,   138,   918.4,      5.7,   13.8,    33,    2755},  // spinach
        { 9.6,    138,   2.7,    54,   290.7,      8.4,    5.4,    83,    1912},  // sweet potatos
        { 8.5,     87,   1.7,   173,    86.8,      1.2,    4.3,    55,      57},  // peaches
        {12.8,     99,   2.5,   154,    85.7,      3.9,    4.3,    65,     257},  // prunes
        {17.4,   1055,   3.7,   459,     5.1,     26.9,   38.2,    93,       0},  // limabeans
        {26.9,   1691,  11.4,   792,     0,       38.4,   24.6,   217,      0} };// navybeans

   double[] daily_allowance =
     {  3.,      70.,    .8,    12.,    5.,       1.8,    2.7,   18.,     75. };
   diet M = new diet("Stinglers Diet Problem",
                        foods,
                        nutrients,
                        daily_allowance,
                        nutritive_value);
   try
   {
     M.solve();
     try
     {
       System.out.println("Solution:");
       double[] x = M.dailyPurchase();
       for (int i = 0; i < x.length; ++i)
           System.out.println(foods[i] + " : $" + x[i]);

       System.out.println("Nutrients:");
       double[] y = M.dailyNutrients();
       for (int i = 0; i < y.length; ++i)
         System.out.println(nutrients[i] + " : " + y[i] + " " + nutrient_unit[i] +
                            " (" + daily_allowance[i] + ")");
     }
     catch (SolutionError e)
     {
       System.out.println("Solution error : " + e.getMessage());
     }
   }
   finally
   {
     M.dispose();
   }
  }
}
```

4.2   Portfolio optimization

4.2.1   The basic model

The classical Markowitz portfolio optimization problem considers investing in n stocks or assets held over a period of time. Let x_j denote the amount invested in asset j, and assume a stochastic model where the return of the assets is a random variable r with known mean

$$\mu = \mathbf{E}r$$

and covariance

$$\Sigma = \mathbf{E}(r - \mu)(r - \mu)^T.$$

The return of the investment is also a random variable $y = r^T x$ with mean (or expected return)

$$\mathbf{E}y = \mu^T x$$

and variance (or risk)

$$(y - \mathbf{E}y)^2 = x^T \Sigma x.$$

The problem facing the investor is to rebalance the portfolio to achieve a good compromise between risk and expected return, e.g., maximize the expected return subject to a budget constraint and an upper bound (denoted γ) on the tolerable risk. This leads to the optimization problem

$$
\begin{array}{lrcl}
\text{maximize} & \mu^T x & & \\
\text{subject to} & e^T x & = & w + e^T x^0, \\
& x^T \Sigma x & \leq & \gamma^2, \\
& x & \geq & 0.
\end{array}
\tag{4.2}
$$

The variables x denotes the investment i.e. x_j is the amount invested in asset j and x_j^0 is the initial holding of asset j. Finally, w is the initial amount of cash available.

A popular choice is $x^0 = 0$ and $w = 1$ because then x_j may be interpretated as the relative amount of the total portfolio that is invested in asset j.

Since e is the vector of all ones then

$$e^T x = \sum_{j=1}^{n} x_j$$

is the total investment. Clearly, the total amount invested must be equal to the initial wealth, which is

$$w + e^T x^0.$$

This leads to the first constraint

$$e^T x = w + e^T x^0.$$

The second constraint

$$x^T \Sigma x \leq \gamma^2$$

ensures that the variance, or the risk, is bounded by the parameter γ^2. Therefore, γ specifies an upper bound of the standard deviation the investor is willing to undertake. Finally, the constraint

$$x_j \geq 0$$

excludes the possibility of short-selling. This constraint can of course be excluded if short-selling is allowed.

The covariance matrix Σ is positive semidefinite by definition and therefore there exist a matrix G such that

$$\Sigma = GG^T. \tag{4.3}$$

In general the choice of G is **not** unique and one possible choice of G is the Cholesky factorization of Σ. However, in many cases another choice is better for efficiency reasons as discussed in Section 4.2.3. For a given G we have that

$$\begin{aligned} x^T \Sigma x &= x^T GG^T x \\ &= \left\| G^T x \right\|^2. \end{aligned}$$

Hence, we may write the risk constraint as

$$\gamma \geq \left\| G^T x \right\|$$

or equivalently

$$[\gamma; G^T x] \in Q^{n+1}.$$

where Q^{n+1} is the $n+1$ dimensional quadratic cone.

Therefore, problem (4.2) can be written as

$$\begin{array}{lrcl} \text{maximize} & \mu^T x \\ \text{subject to} & e^T x & = & w + e^T x^0, \\ & [\gamma; G^T x] & \in & Q^{n+1}, \\ & x & \geq & 0, \end{array} \tag{4.4}$$

which is a conic quadratic optimization problem that can easily be formulated and solved with Fusion.

Subsequently we will use the example data

$$\mu = \begin{bmatrix} 0.1073 \\ 0.0737 \\ 0.0627 \end{bmatrix}$$

and

$$\Sigma = 0.1 \begin{bmatrix} 0.2778 & 0.0387 & 0.0021 \\ 0.0387 & 0.1112 & -0.0020 \\ 0.0021 & -0.0020 & 0.0115 \end{bmatrix}.$$

This implies

$$G^T = \sqrt{0.1} \begin{bmatrix} 0.5271 & 0.0734 & 0.0040 \\ 0 & 0.3253 & -0.0070 \\ 0 & 0 & 0.1069 \end{bmatrix}$$

using 5 figures of accuracy. The data is taken from [1]. Using the code described in the following section the result

```
Expected return 6.8475e-02 for std. deviation: 3.5000e-02
Expected return 7.0998e-02 for std. deviation: 4.0000e-02
Expected return 7.4781e-02 for std. deviation: 5.0000e-02
Expected return 7.8039e-02 for std. deviation: 6.0000e-02
Expected return 8.1084e-02 for std. deviation: 7.0000e-02
Expected return 8.4016e-02 for std. deviation: 8.0000e-02
Expected return 8.6880e-02 for std. deviation: 9.0000e-02
```

is obtained. It can be seen that as the expected return increases then a higher risk must also be accepted as expected. The code described in this section is an excerpt from the portfolio example available in the distribution as `portfolio.java` and located in the folder

```
C:\Program Files\mosek\7\tools\examples\fusion\java
```

for Windows installation or in

```
$HOME/mosek/7/tools/examples/fusion/java
```

for Unix systems. The program requires the user to provide input data by a set of files in CSV format, all sharing the same first part of their name as follows:

file name	data type	data storage info
`rootname-n.txt`	the number of assets n	single value
`rootname-gammas.csv`	the set of standard deviations γ	column
`rootname-GT.csv`	the factorized covariance matrix G^T	a row per line
`rootname-mu.csv`	the expected returns μ	column
`rootname-w.csv`	the initial available cash w	column
`rootname-x0.csv`	the initial portfolio x_0	column
`rootname-as.csv`	the set of weights α to compute the efficient frontier	column

Note that `rootname-n.txt` has a different extension to mark that it contains the problem dimension. The input files for this case of study have `rootname` equal to `portex` and they are located in the folder

```
C:\Program Files\mosek\7\tools\examples\fusion\data
```

for Windows installation or in

```
$HOME/mosek/7/tools/examples/fusion/data
```

for Unix systems. The program `portfolio.java` is executed passing `path-to-data/portex` as command line parameter.

4.2.1.1 Example code

The following example code demonstrates how the basic Markowitz model (4.4) is implemented using Fusion.

```java
/*
Purpose:
    Computes the optimal portfolio for a given risk

Input:
    n: Number of assets
    mu: An n dimmensional vector of expected returns
    GT: A matrix with n columns so (GT')*GT  = covariance matrix
    x0: Initial holdings
    w: Initial cash holding
    gamma: Maximum risk (=std. dev) accepted

Output:
    Optimal expected return and the optimal portfolio
*/
public static double BasicMarkowitz
  ( int n,
    double[] mu,
    DenseMatrix GT,
    double[] x0,
    double   w,
    double   gamma)
  throws mosek.fusion.SolutionError
{

  Model M = new Model("Basic Markowitz");
  try
   {
      // Redirect log output from the solver to stdout for debugging.
      // if uncommented.
      //M.setLogHandler(new java.io.PrintWriter(System.out));

      // Defines the variables (holdings). Shortselling is not allowed.
      Variable x = M.variable("x", n, Domain.greaterThan(0.0));

      //  Maximize expected return
      M.objective("obj", ObjectiveSense.Maximize, Expr.dot(mu,x));

      // The amount invested  must be identical to intial wealth
      M.constraint("budget", Expr.sum(x), Domain.equalsTo(w+sum(x0)));

      // Imposes a bound on the risk
      M.constraint("risk", Expr.vstack(Expr.constTerm(new double[] {gamma}),Expr.mul(GT,x)), Domain.inQCone());

      // Solves the model.
```

```
        M.solve();

        return dot(mu,x.level());
    }
  finally
  {
      M.dispose();
  }
}
```

The source code should be self-explanatory except perhaps for

```
 M.constraint("risk", Expr.vstack(Expr.constTerm(new double[] {gamma}),Expr.mul(GT,x)), Domain.inQCone());
```

where the linear expression

$$\begin{bmatrix} \gamma \\ G^T x \end{bmatrix}$$

is created using the vstack operator. Finally, the linear expression must lie in a quadratic cone implying

$$\gamma \geq \left\| G^T x \right\|.$$

4.2.2   The efficient frontier

The portfolio computed by the Markowitz model is efficient in the sense that there is no other portfolio which gives a strictly higher return for the same amount of risk. An efficient portfolio is also sometimes called a Pareto optimal portfolio. Clearly, an investor should only invest in efficient portfolios and therefore it may be relevant to present the investor all efficient portfolios so the investor can choose the portfolio that has the desired tradeoff between return and risk.

Given a nonnegative α then the problem

$$\begin{array}{lll} \text{maximize} & \mu^T x - \alpha s \\ \text{subject to} & e^T x & = & w + e^T x^0, \\ & [s; G^T x] & \in & Q^{n+1}, \\ & x & \geq & 0. \end{array} \tag{4.5}$$

computes an efficient portfolio. Note that the objective maximize the expected return while maximizing $-\alpha$ times the standard deviation. Hence, the standard deviation is minimized while α specifies the tradeoff between expected return and risk.

Ideally the problem 4.5 should be solved for all values $\alpha \geq 0$ but in practice impossible. Using the example data from Section 4.2.1, the optimal values of return and risk for several αs are listed below:

```
Efficient frontier
alpha           return          risk
0.0000          1.0730e-01      7.2700e-01
0.0100          1.0730e-01      1.6667e-01
0.1000          1.0730e-01      1.6667e-01
0.2500          1.0321e-01      1.4974e-01
0.3000          8.0529e-02      6.8144e-02
0.3500          7.4290e-02      4.8585e-02
```

```
0.4000       7.1958e-02    4.2309e-02
0.4500       7.0638e-02    3.9185e-02
0.5000       6.9759e-02    3.7327e-02
0.7500       6.7672e-02    3.3816e-02
1.0000       6.6805e-02    3.2802e-02
1.5000       6.6001e-02    3.2130e-02
2.0000       6.5619e-02    3.1907e-02
3.0000       6.5236e-02    3.1747e-02
10.0000      6.4712e-02    3.1633e-02
```

4.2.2.1 Example code

The following example code demonstrates how to compute the efficient portfolios for several values of α in Fusion.

```java
/*
  Purpose:
      Computes several portfolios on the optimal portfolios by

          for alpha in alphas:
              maximize    expected return - alpha * standard deviation
              subject to the constraints

  Input:
      n: Number of assets
      mu: An n dimmensional vector of expected returns
      GT: A matrix with n columns so (GT')*GT  = covariance matrix
      x0: Initial holdings
      w: Initial cash holding
      alphas: List of the alphas

  Output:
      The efficient frontier as list of tuples (alpha,expected return,risk)
*/
public static void EfficientFrontier
  ( int n,
    double[] mu,
    DenseMatrix GT,
    double[]    x0,
    double      w,
    double[]    alphas,

    double[]    frontier_mux,
    double[]    frontier_s)
  throws mosek.fusion.SolutionError
{

  Model M = new Model("Efficient frontier");
  try
  {
      //M.setLogHandler(new java.io.PrintWriter(System.out));

      // Defines the variables (holdings). Shortselling is not allowed.
      Variable x = M.variable("x", n, Domain.greaterThan(0.0)); // Portfolio variables
      Variable s = M.variable("s", 1, Domain.unbounded()); // Risk variable

      M.constraint("budget", Expr.sum(x), Domain.equalsTo(w+sum(x0)));
```

```
        // Computes the risk
        M.constraint("risk", Expr.vstack(s.asExpr(),Expr.mul(GT,x)),Domain.inQCone());

        for (int i = 0; i < alphas.length; ++i)
            {
                //  Define objective as a weighted combination of return and risk
                M.objective("obj", ObjectiveSense.Maximize, Expr.sub(Expr.dot(mu,x),Expr.mul(alphas[i],s)));

                M.solve();

                frontier_mux[i] = dot(mu,x.level());
                frontier_s[i]   = s.level()[0];
            }
    }
    finally
    {
        M.dispose();
    }
}
```

Note the efficient frontier could also have been computed using the code in Section 4.2.1.1 by varying γ. However, when the constraints of a Fusion model is changed the model has to be rebuild whereas a rebuild is not needed if only the objective is modified.

4.2.3   Improving the computational efficiency

In practice it is often important to solve the portfolio problem in a short amount of time. Therefore, in this section it is discussed what can be done at the modelling stage to improve the computational efficiency.

The computational cost is of course to some extent dependent on the number of constraints and variables in the optimization problem. However, in practice a more important factor is the number of nonzeros used to represent the problem. Indeed it is often better to focus on the number of nonzeros in G (see (4.3)) and try to reduce that number by for instance changing the choice of G.

In other words if the computational efficiency should be improved then it is always a good idea to start with focusing at the covariance matrix. As an example assume that

$$\Sigma = D + VV^T$$

where D is a positive definite diagonal matrix. Moreover, V is a matrix with n rows and p columns. Such a model for the covariance matrix is called a factor model and usually p is much smaller than n. In practice p tends be a small number independent of n say less than 100.

One possible choice for G is the Cholesky factorization of Σ which requires storage proportional to $n(n+1)/2$. However, another choice is

$$G^T = \left[\begin{array}{c} D^{1/2} \\ V^T \end{array} \right]$$

because then

$$GG^T = D + VV^T.$$

This choice requires storage proportional to $n + pn$ which is much less than for the Cholesky choice of G. Indeed assuming p is a constant then the difference in storage requirements is a factor of n.

The example above exploits the so-called factor structure and demonstrates that an alternative choice of G may lead to a significant reduction in the amount of storage used to represent the problem. This will in most cases also lead to a significant reduction in the solution time.

The lesson to be learned is that it is important to investigate how the covariance is formed. Given this knowledge it might be possible to make a special choice for G that helps reducing the storage requirements and enhance the computational efficiency.

4.2.4 Slippage cost

The basic Markowitz model assumes that there are no costs associated with trading the assets and that the returns of the assets is independent of the amount traded. None of those assumptions are usually valid in practice. Therefore, a more realistic model is

$$
\begin{aligned}
\text{maximize} \quad & \mu^T x \\
\text{subject to} \quad & e^T x + \sum_{j=1}^{n} T_j(x_j - x_j^0) && = && w + e^T x^0, \\
& x^T \Sigma x && \leq && \gamma^2, \\
& x && \geq && 0,
\end{aligned}
\tag{4.6}
$$

where the function

$$T_j(x_j - x_j^0)$$

specifies the transaction costs when the holding of asset j is changed from its initial value.

4.2.4.1 Market impact costs

If the initial wealth is fairly small and no short selling is allowed, then the holdings will be small and then the amount traded of each asset must also be small. Therefore, it is reasonable to assume that the prices of the assets is independent of the amount traded. However, if a large volume of an assert is sold or purchased it can be expected that the price changes and hence the expected return also changes. This effect is called market impact costs. It is common to assume that the market impact cost for asset j can be modelled by

$$m_j \sqrt{|x_j - x_j^0|}$$

according where m_j is a constant that is estimated in some way. See [2][p. 452] for details. Hence, we have

$$T_j(x_j - x_j^0) = m_j|x_j - x_j^0|\sqrt{|x_j - x_j^0|} = m_j|x_j - x_j^0|^{3/2}.$$

From [3] it is known that

$$\{(t, z) : t \geq z^{3/2}, z \geq 0\} = \{(t, z) : (s, t, z), (z, 1/8, s) \in Q_r^3\}$$

where Q_r^3 is the 3 dimensional rotated quadratic cone. Hence, it follows

$$
\begin{array}{rcl}
z_j & = & |x_j - x_j^0|, \\
(s_j, t_j, z_j), (z_j, 1/8, s_j) & \in & Q_r^3, \\
\displaystyle\sum_{j=1}^n T(x_j - x_j^0) & = & \displaystyle\sum_{j=1}^n t_j.
\end{array}
$$

Unfortunately this set of constraints is nonconvex due to the constraint

$$z_j = |x_j - x_j^0| \tag{4.7}$$

but in many cases the constraint may be replaced by the relaxed constraint

$$z_j \geq |x_j - x_j^0|. \tag{4.8}$$

which is equivalent to

$$
\begin{array}{rcl}
z_j & \geq & x_j - x_j^0, \\
z_j & \geq & -(x_j - x_j^0).
\end{array}
\tag{4.9}
$$

For instance if the universe of assets contains a risk free asset then

$$z_j > |x_j - x_j^0| \tag{4.10}$$

cannot hold for an optimal solution.

Now given that the optimal solution has the property that (4.10) holds then the market impact costs within the model is larger than the true market impact cost and hence money are essentially considered garbage and removed by generating transaction costs. This may happen if a portfolio with very small risk is requested because then the only way to obtain a small risk is to get rid of some of the assets by generating transaction costs. It is assumed this is not the case and hence the models (4.7) and (4.8) are equivalent.

The above observations leads to

$$
\begin{array}{llll}
\text{maximize} & \mu^T x \\
\text{subject to} & e^T x + m^T t & = & w + e^T x^0, \\
& (\gamma, G^T x) & \in & Q^{n+1}, \\
& z_j & \geq & x_j - x_j^0, & j = 1, \ldots, n, \\
& z_j & \geq & x_j^0 - x_j, & j = 1, \ldots, n, \\
& [v_j; t_j; z_j], [z_j; 1/8; v_j] & \in & Q_r^3, & j = 1, \ldots, n, \\
& x & \geq & 0.
\end{array}
$$

The revised budget constraint

$$e^T x = w + e^T x^0 - m^T t$$

specifies that the total investment must be equal to the initial wealth minus the transaction costs. Moreover, observe the variables v and z are some auxiliary variables that model the market impact cost. Indeed it holds

$$z_j \geq |x_j - x_j^0|$$

and

$$t_j \geq z_j^{3/2}.$$

Before proceeding it should be mentioned that transaction costs of the form

$$c_j \geq z_j^{p/q}$$

where p and q are both integers and $p \geq q$ can be modelled using quadratic cones. See [3] for details.

Example code

The following example code demonstrates how to compute an optimal portfolio when market impact cost are included using Fusion.

```
/*
    Description:
        Extends the basic Markowitz model with a market cost term.

    Input:
        n: Number of assets
        mu: An n dimmensional vector of expected returns
        GT: A matrix with n columns so (GT')*GT  = covariance matrix'
        x0: Initial holdings
        w: Initial cash holding
        gamma: Maximum risk (=std. dev) accepted
        m: It is assumed that  market impact cost for the j'th asset is
            m_j|x_j-x0_j|^3/2

    Output:
        Optimal expected return and the optimal portfolio

*/
public static void MarkowitzWithMarketImpact
  ( int n,
    double[] mu,
    DenseMatrix GT,
    double[] x0,
    double   w,
    double   gamma,
    double[] m,

    double[] xsol,
```

```java
      double[] tsol)
    throws mosek.fusion.SolutionError
{
  Model M = new Model("Markowitz portfolio with market impact");
  try
  {
      //M.setLogHandler(new java.io.PrintWriter(System.out));

      // Defines the variables. No shortselling is allowed.
      Variable x = M.variable("x", n, Domain.greaterThan(0.0));

      // Addtional "helper" variables
      Variable t = M.variable("t", n, Domain.unbounded());
      Variable z = M.variable("z", n, Domain.unbounded());
      Variable v = M.variable("v", n, Domain.unbounded());

      //  Maximize expected return
      M.objective("obj", ObjectiveSense.Maximize, Expr.dot(mu,x));

      // Invested amount + slippage cost = initial wealth
      M.constraint("budget", Expr.add(Expr.sum(x),Expr.dot(m,t)), Domain.equalsTo(w+sum(x0)));

      // Imposes a bound on the risk
      M.constraint("risk", Expr.vstack(Expr.constTerm(new double[] {gamma}),Expr.mul(GT,x)),
                   Domain.inQCone());

      // z >= |x-x0|
      M.constraint("buy", Expr.sub(z,Expr.sub(x,x0)),Domain.greaterThan(0.0));
      M.constraint("sell", Expr.sub(z,Expr.sub(x0,x)),Domain.greaterThan(0.0));

      // t >= z^1.5, z >= 0.0. Needs two rotated quadratic cones to model this term
      M.constraint("ta", Expr.hstack(v.asExpr(),t.asExpr(),z.asExpr()),Domain.inRotatedQCone());
      M.constraint("tb", Expr.hstack(z.asExpr(),Expr.constTerm(n,1.0/8.0),v.asExpr()),
                   Domain.inRotatedQCone());

      M.solve();

      if (xsol != null)
          System.arraycopy(x.level(),0,xsol,0,n);
      if (tsol != null)
          System.arraycopy(t.level(),0,tsol,0,n);
  }
  finally
  {
      M.dispose();
  }
}
```

The major new feature compared to the previous examples are

```java
 M.constraint("ta", Expr.hstack(v.asExpr(),t.asExpr(),z.asExpr()),Domain.inRotatedQCone());
```

and

```java
 M.constraint("tb", Expr.hstack(z.asExpr(),Expr.constTerm(n,1.0/8.0),v.asExpr()),
        Domain.inRotatedQCone());
```

In the first line the variables v, t and z are stacked horizontally which corresponds to creating a list of linear expressions where the j'th element has the form

$$\begin{bmatrix} v_j \\ t_j \\ z_j \end{bmatrix}$$

and finally each linear expression is constrained to be in rotated quadratic cone i.e.

$$2v_j t_j \geq z_j^2 \text{ and } v_j, t_j \geq 0.$$

Similarly the second line is equivalent to the constraint

$$\begin{bmatrix} z_j \\ 1/8 \\ v_j \end{bmatrix} \in Q_r^3$$

or equivalently

$$2z_j \frac{1}{8} \geq v_j^2 \text{ and } z_j \geq 0.$$

4.2.4.2 Transaction costs

Now assume there is a cost associated with trading asset j and the cost is given by

$$T_j(\Delta x_j) = \begin{cases} 0, & \Delta x_j = 0, \\ f_j + g_j|\Delta x_j|, & \text{otherwise.} \end{cases}$$

Δx_j is the change in the holding of asset j i.e.

$$\Delta x_j = x_j - x_j^0.$$

Hence, whenever asset j is traded a fixed cost of f_j has to be paid and a variable cost of g_j per unit traded. Given the assumptions about transaction costs in this section then problem (4.6) may be formulated as

$$\begin{aligned}
\text{maximize} \quad & \mu^T x \\
\text{subject to} \quad & e^T x + \sum_{j=1}^n (f_j y_j + g_j z_j) & = \quad & w + e^T x^0, \\
& [\gamma; G^T x] & \in \quad & Q^{n+1}, \\
& z_j & \geq \quad & x_j - x_j^0, \quad j = 1, \dots, n, \\
& z_j & \geq \quad & x_j^0 - x_j, \quad j = 1, \dots, n, \\
& z_j & \leq \quad & U_j y_j, \quad j = 1, \dots, n, \\
& y_j & \in \quad & \{0, 1\}, \quad j = 1, \dots, n, \\
& x & \geq \quad & 0.
\end{aligned}$$

First observe that

$$z_j \geq |x_j - x_j^0|$$

and hence z_j is bounded below by $|\Delta x_j|$. U_j is some a prior chosen upper bound on the amount of trading in asset j and therefore if $z_j > 0$ then $y_j = 1$ has to be the case. This implies that the transaction costs for the asset j is given by

$$f_j y_j + g_j z_j.$$

Example code

The following example code demonstrates how to compute an optimal portfolio when transaction costs are included.

```java
/*
    Description:
        Extends the basic Markowitz model with a market cost term.

    Input:
        n: Number of assets
        mu: An n dimmensional vector of expected returns
        GT: A matrix with n columns so (GT')*GT  = covariance matrix
        x0: Initial holdings
        w: Initial cash holding
        gamma: Maximum risk (=std. dev) accepted
        f: If asset j is traded then a fixed cost f_j must be paid
        g: If asset j is traded then a cost g_j must be paid for each unit traded

    Output:
        Optimal expected return and the optimal portfolio

*/
public static double[] MarkowitzWithTransactionsCost
  ( int n,
    double[] mu,
    DenseMatrix GT,
    double[] x0,
    double    w,
    double    gamma,
    double[] f,
    double[] g)
  throws mosek.fusion.SolutionError
{

    // Upper bound on the traded amount
    double[] u = new double[n];
    {
      double v = w+sum(x0);
      for (int i = 0; i < n; ++i) u[i] = v;
    }

    Model M = new Model("Markowitz portfolio with transaction costs");
    try
    {
        //M.setLogHandler(new java.io.PrintWriter(System.out));
```

```
// Defines the variables. No shortselling is allowed.
Variable x = M.variable("x", n, Domain.greaterThan(0.0));

// Addtional "helper" variables
Variable z = M.variable("z", n, Domain.unbounded());
// Binary varables
Variable y = M.variable("y", n, Domain.inRange(0.0,1.0), Domain.isInteger());

//  Maximize expected return
M.objective("obj", ObjectiveSense.Maximize, Expr.dot(mu,x));

// Invest amount + transactions costs = initial wealth
M.constraint("budget", Expr.add(Expr.add(Expr.sum(x),Expr.dot(f,y)),Expr.dot(g,z)),
             Domain.equalsTo(w+sum(x0)));

// Imposes a bound on the risk
M.constraint("risk", Expr.vstack(Expr.constTerm(new double[] {gamma}),Expr.mul(GT,x)),
             Domain.inQCone());

// z >= |x-x0|
M.constraint("buy", Expr.sub(z,Expr.sub(x,x0)),Domain.greaterThan(0.0));
M.constraint("sell", Expr.sub(z,Expr.sub(x0,x)),Domain.greaterThan(0.0));

//M.constraint("trade", Expr.hstack(z.asExpr(),Expr.sub(x,x0)), Domain.inQcone())"

// Consraints for turning y off and on. z-diag(u)*y<=0 i.e. z_j <= u_j*y_j
M.constraint("y_on_off", Expr.sub(z,Expr.mul(Matrix.diag(u),y)), Domain.lessThan(0.0));

// Integer optimization problems can be very hard to solve so limiting the
// maximum amount of time is a valuable safe guard
M.setSolverParam("mioMaxTime", 180.0);
M.solve();

return x.level();
    }
  finally
  {
    M.dispose();
  }
}
```

4.3 Traffic Network

This example is base on a presentation by Robert Fourer [4].

The traffic network is defined as a simple network flow problem, where each arc represents a road, and each node an intersection. The flow represents the amount of traffic; more traffic will decrease the transport time through an arc as it approaches the capacity for the arc. We wish to minimize the overage travel time from source to sink in the network for a given amount of traffic.

This problem can be solved as a conic quadratic problem.

We define the network from the following data:

- A set of nodes N

- A source e

- A sink f

- A set of (directed) arcs $A \subseteq N \cup e \times N \cup f$

and associated parameters

$$
\begin{array}{ll}
b_{ij} & \text{Base travel time for arc } (i, j) \in A \\
c_{ij} & \text{Capacity of arc } (i, j) \in A \\
s_{ij} & \text{Traffic sensitivity for arc } (i, j) \in A \\
T & \text{Desired flow from } e \text{ to } f
\end{array}
$$

We operate with two sets of variables:

$$
\begin{array}{ll}
x_{ij} & \text{Flow through arc } (i, j) \in A \\
t_{ij} & \text{Travel time through arc } (i, j) \in A
\end{array}
$$

We can now write the model as

$$
\begin{aligned}
\text{minimize} \quad & \sum_{(i,j) \in A} t_{ij} x_{ij} / T & (0) \\
\text{subject to} \quad & t_{ij} = b_{ij} + \frac{s_{ij} x_{ij}}{1 - x_{ij}/c_{ij}} \text{ for } (i, j) \in A & (1) \\
& \sum_{j:(i,j) \in A} (x_{ij}) = \sum_{j:(j,i) \in A} (x_{ji}) \text{ for } i \in N & (2) \\
& \sum_{j:(e,j) \in A} x_{ej} = T & (3) \\
& 0 \leq x_{ij} \leq c_{ij} \text{ for } (i, j) \in A & (4)
\end{aligned}
$$

This model is clearly non-linear.

The objective (0) is to minimize the overall average travel time.

The first constraint (1) describes how the travel-time increases for a node as the flow approaches capacity. This puts an implicit bound on the flow since the $t_{ij} \to \infty$ as $x_{ij} \to c_{ij}$.

The second constraint (2) demands that inflow must equal outflow for each node.

The third constraint (3) defines the total inflow through the source. Implicitly this must be equal to the total outflow through the sink.

The last constraint (4) demands that all flows are non-negative. By construction, this implies that all travel times t_{ij} are positive.

4.3.1   Reformulating as conic convex model

We have two non-linear elements; the objective and constraint (1). First of all, consider the objective

$$\text{minimize} \sum_{(i,j) \in A} t_{ij} x_{ij} / T$$

If we substitute t_{ij} from constraint (1) into this we get that the terms of the sum of the objective can be written as

$$t_{ij} x_{ij} = x_{ij} b_{ij} + s_{ij} \frac{x_{ij}^2}{1 - x_{ij}/c_{ij}}$$

Consider the term

$$s_{ij} \frac{x_{ij}^2}{1 - x_{ij}/c_{ij}}.$$

This will always be positive for $x_{ij} < c_{ij}$. In other words, if we add a variable vector d_{ij}, such that

$$d_{ij} \geq s_{ij} \frac{x_{ij}^2}{1 - x_{ij}/c_{ij}}$$

we may write out problem as

$$
\begin{aligned}
\text{minimize} \quad & \sum_{(i,j) \in A} (x_{ij} b_{ij} + d_{ij})/T && \text{(0a)} \\
\text{subject to} \quad & d_{ij} \geq s_{ij} \frac{x_{ij}^2}{1 - x_{ij}/c_{ij}} \text{ for } (i,j) \in A && \text{(1a)} \\
& \sum_{j:(i,j) \in A} (x_{ij}) = \sum_{j:(j,i) \in A} (x_{ji}) \text{ for } i \in N && \text{(2)} \\
& \sum_{(e,j) \in A} x_{ej} = T && \text{(3)} \\
& 0 \leq x_{ij} \leq c_{ij} \text{ for } (i,j) \in A && \text{(4)}
\end{aligned}
$$

We now have only constraint (1a) to worry about. We can write (1a) equivalently as

$$d_{ij}(1 - x_{ij}/c_{ij})/s_{ij} \geq x_{ij}^2$$

Again, we notice that the part $(1 - x_{ij}/c_{ij})/s_{ij}$ is always positive.

We add yet another set of variables, z_{ij} and replace (1a) by two sets of constraints:

$$
\begin{aligned}
d_{ij} z_{ij} &\geq x_{ij}^2 \text{ for } (i,j) \in A && \text{(1b)} \\
z_{ij} &= (1 - x_{ij}/c_{ij})/s_{ij} \text{ for } (i,j) \in A && \text{(1c)}
\end{aligned}
$$

Here, (1c) is linear and (1b) is a rotated quadratic cone.

Out final model the looks like this:

$$\text{minimize} \quad \sum_{(i,j)\in A} (x_{ij}b_{ij} + d_{ij})/T \qquad\qquad\qquad (0\text{a})$$

$$\text{subject to} \quad 2d_{ij}z_{ij} \geq x_{ij}^2 \text{ for } (i,j) \in A \qquad\qquad (1\text{a})$$

$$z_{ij} = \frac{1}{2}(1 - x_{ij}/c_{ij})/s_{ij} \text{ for } (i,j) \in A \qquad (1\text{b})$$

$$\sum_{j:(i,j)\in A} (x_{ij}) + x_{ej} = \sum_{j:(j,i)\in A} (x_{ji}) + x_{jf} \text{ for } i \in N \quad (2)$$

$$\sum_{(e,j)\in A} x_{ej} = T \qquad\qquad\qquad\qquad (3)$$

$$x_{ij}, d_{ij}, z_{ij} \geq 0 \text{ for } (i,j) \in A \qquad\qquad (4)$$

This is a conic quadratic model that can be solved with MOSEK.

4.3.2　Fusion model for traffic network

We now wish to put the model on a form that can be implemented in Fusion. First of all, let n denote the number of nodes, including e and f. The set A denotes *all* arcs, including those to the sink and from the source. Finally we introduce a symbolic arc (f, e) with a fixed capacity and flow of T — this arc will have no influence on the objective.

We define the constants for base travel time, capacity and sensitivity $b, c, s \in \mathbb{R}^{n\times n}$ as two-dimensional matrices, and we use the convention that $b' \in \mathbb{R}^{n\times n}$ subject to

$$b'_{ij} = \begin{cases} b_{ij}^{-1} \text{ for } (i,j) \in A \\ 0 \text{ otherwise} \end{cases}$$

and similarly for c' and s'. Furthermore, we introduce the constant $\mathbf{e} \in \mathbb{R}^{n\times n}$

$$\mathbf{e}_{ij} = \begin{cases} 1 \text{ for } (i,j) \in A \\ 0 \text{ otherwise} \end{cases}$$

and $\mathbf{e}_e \in \mathbb{R}^{n\times n}$ with $\mathbf{x}_{fe} = 1$ and all other entries 0.

We can now define the model on vector form:

$$\text{minimize} \quad \frac{1}{T}(x \cdot b + \mathbf{e} \cdot d) \qquad\qquad\qquad (0\text{a})$$

$$\text{subject to} \quad 2d_{ij}z_{ij} \geq x_{ij}^2 \text{ for } (i,j) \in A \qquad (1\text{a})$$

$$z + \frac{1}{2}c' \circ s' \circ x = \frac{1}{2}s' \qquad\qquad (1\text{b})$$

$$\text{diag}\left((\mathbf{e}+\mathbf{e}_e)^T x - x^T(\mathbf{e}+\mathbf{e}_e)\right) = 0 \quad (2)$$

$$x_{fe} = T \qquad\qquad\qquad\qquad (3)$$

$$x, d, z \in \mathbb{R}^{n\times n}, \ 0 \leq x \leq c, d, z \geq 0 \qquad (4\text{a})$$

where $\text{diag}(m)$ is the diagonal of the square matrix m as a vector. The operator $\circ$ means *element-wise multiplication*.

We will use the following network as data for the implementation:
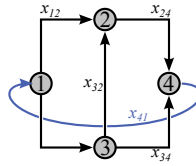
Figure 4.1: Very simple traffic network example. Source: Robert Fourer, Convexity Detection in Large-Scale Optimization.

4.3.3   Source code

```
//
// Copyright: Copyright (c) MOSEK ApS, Denmark. All rights reserved.
//
// File:       TrafficNetworkModel.java
//
// Purpose:    Demonstrates a traffic network problem as a conic quadratic problem.
//
// Source:     Robert Fourer, "Convexity Checking in Large-Scale Optimization",
//             OR 53 --- Nottingham 6-8 September 2011.
//
// The problem:
//             Given a directed graph representing a traffic network
//             with one source and one sink, we have for each arc an
//             associated capacity, base travel time and a
//             sensitivity. Travel time along a specific arc increases
//             as the flow approaches the capacity.
//
//             Given a fixed inflow we now wish to find the
//             configuration that minimizes the average travel time.

package com.mosek.fusion.examples;
import mosek.fusion.*;

public class TrafficNetworkModel extends Model
{
    public class TrafficNetworkError extends Exception
    {
        public TrafficNetworkError(String msg) { super(msg); }
    }

    private Variable flow;

    public TrafficNetworkModel
        ( int       numberOfNodes,
          int       source_idx,
          int       sink_idx,
          int[]     arc_i,
          int[]     arc_j,
          double[] arcSensitivity,
          double[] arcCapacity,
          double[] arcBaseTravelTime,
          double    T)
```

```
{
    super("Traffic Network");
    boolean finished = false;
    try
    {
      int n = numberOfNodes;
      int narcs = arc_j.length;

      double[] n_ones = new double[narcs]; for (int i = 0; i < narcs; ++i) n_ones[i] = 1.0;
      Set NxN = new NDSet(new int[] { 0, 0 },
                          new int[] { n, n});
      Matrix sens =
          Matrix.sparse(n, n, arc_i, arc_j, arcSensitivity);
      Matrix cap =
          Matrix.sparse(n, n, arc_i, arc_j, arcCapacity);
      Matrix basetime =
          Matrix.sparse(n, n, arc_i, arc_j, arcBaseTravelTime);
      Matrix e =
          Matrix.sparse(n, n, arc_i, arc_j, n_ones);
      Matrix e_e =
          Matrix.sparse(n,n,
                        new int[]  { sink_idx}, new int[] { source_idx},
                        new double[] { 1.0 });

      double[] cs_inv = new double[narcs];
      double[] s_inv  = new double[narcs];
      for (int i = 0; i < narcs; ++i) cs_inv[i] = 1.0 / (arcSensitivity[i] * arcCapacity[i]);
      for (int i = 0; i < narcs; ++i) s_inv[i]  = 1.0 / arcSensitivity[i];
      Matrix cs_inv_matrix = Matrix.sparse(n, n, arc_i, arc_j, cs_inv);
      Matrix s_inv_matrix  = Matrix.sparse(n, n, arc_i, arc_j, s_inv);

      flow       = variable("traffic_flow", NxN, Domain.greaterThan(0.0));

      Variable x = flow;
      Variable t = variable("travel_time" , NxN, Domain.greaterThan(0.0));
      Variable d = variable("d",              NxN, Domain.greaterThan(0.0));
      Variable z = variable("z",              NxN, Domain.greaterThan(0.0));


      // Set the objective:
      objective("Average travel time",
                ObjectiveSense.Minimize,
                Expr.mul(1.0/T, Expr.add(Expr.dot(basetime,x), Expr.dot(e,d))));

      // Set up constraints
      // Constraint (1a)
      {
          Variable[][] v = new Variable[narcs][3];
          for (int i = 0; i < narcs; ++i) {
                  v[i][0] = d.index(arc_i[i],arc_j[i]);
                  v[i][1] = z.index(arc_i[i],arc_j[i]);
                  v[i][2] = x.index(arc_i[i],arc_j[i]);
              }
          constraint("(1a)",
                      Variable.stack(v),Domain.inRotatedQCone(narcs,3));
      }


      // Constraint (1b)
```

```
            constraint("(1b)",
                      Expr.sub(Expr.add(Expr.mulElm(z,e),
                                        Expr.mulElm(x,cs_inv_matrix)),
                              s_inv_matrix),
                      Domain.equalsTo(0.0));

      // Constraint (2)
      constraint("(2)",
                Expr.sub(Expr.add(Expr.mulDiag(x, e.transpose()),
                                  Expr.mulDiag(x, e_e.transpose())),
                        Expr.add(Expr.mulDiag(x.transpose(), e),
                                 Expr.mulDiag(x.transpose(), e_e))),
                Domain.equalsTo(0.0));
      // Constraint (3)
      constraint("(3)",
                x.index(sink_idx, source_idx), Domain.equalsTo(T));
      finished = true;
    }
    finally
    {
      if (! finished)
      {
        dispose();
      }
    }
  }

  // Return the solution. We do this the easy and inefficeint way:
  // We fetch the whole NxN array og values, a lot of which are
  // zeros.
  public double[] getFlow()
      throws SolutionError
  {
    return flow.level();
  }

  public static void main(String[] args)
      throws SolutionError
  {
    int n            = 4;
    int[]    arc_i   = new int[]    {  0,    0,    2,    1,    2 };
    int[]    arc_j   = new int[]    {  1,    2,    1,    3,    3 };
    double[] arc_base = new double[] {  4.0,  1.0,  2.0,  1.0,  6.0 };
    double[] arc_cap  = new double[] { 10.0, 12.0, 20.0, 15.0, 10.0 };
    double[] arc_sens = new double[] {  0.1,  0.7,  0.9,  0.5,  0.1 };

    double   T          = 20.0;
    int      source_idx = 0;
    int      sink_idx   = 3;

    TrafficNetworkModel M =
        new TrafficNetworkModel(n, source_idx, sink_idx,
                                arc_i, arc_j,
                                arc_sens,
                                arc_cap,
                                arc_base,
                                T);
    try
```
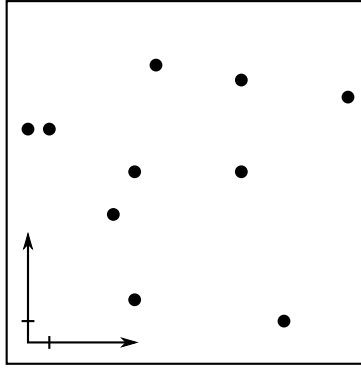
Figure 4.2: Customer locations.

```
    {
      M.solve();

      double[] flow = M.getFlow();

      System.out.println("Optimal flow:");
      for (int i = 0; i < arc_i.length; ++i)
          System.out.println("\tflow node" + arc_i[i] + "->node" + arc_j[i] + " = " + flow[arc_i[i] *
n + arc_j[i]]);
    }
    finally
    {
      M.dispose();
    }
  }
}
```

4.4   Facility location

The simple example above introduces the basic elements of the API, but several important points are not illustrated. This sections presents a more realistic example and explains the details.

The problem is this: Suppose we have a set of locations of customers defined by coordinates in two dimensions (see fig. 4.2). Then we wish to place a facility somewhere such that the total sum of euclidean distances between the customers and the facility is minimized. Formally, this can be written as:

$$\text{minimize} \sum_{i=1}^{N} |p - v_i|$$

where $v_i = (c_i^x, c_i^y)$ is the location of customer i, $p = (x, y)$ is the location of the facility, and N is the number if customers.

To solve this, we need to reformulate it as a conic problem with a linear objective. We do this as follows: First we introduce a set of variables which measures the individual distances between a customer and the facility and redefine the objective

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{N} d_i \\
\text{subject to} \quad & d_i > |p - v_i|, \quad i = 1, \dots, N
\end{aligned}
$$

Then we introduce further variables d_i^x and d_i^y measuring the distance in the x- and y-direction for each customer; then we get the model

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{N} d_i \\
\text{subject to} \quad & d_i > \sqrt{(t_i^x)^2 + (t_i^y)^2}, \quad i = 1, \dots, N \\
& t_i^x = v_i^x - p^x, \qquad\qquad i = 1, \dots, N \\
& t_i^y = v_i^y - p^y, \qquad\qquad i = 1, \dots, N.
\end{aligned}
\tag{4.11}
$$

Here, the objective is linear, the two last sets of constraints are linear and the remaining constraint is conic quadratic; we can implement this model in Fusion. Since the objective minimizes each d_i, we know that at the optimal solution $d_i = \sqrt{(t_i^x)^2 + (t_i^y)^2}$.

We start be defining out data; this is simply the coordinates of the 10 customers shown in figure 4.2:

```
private static Matrix
  customerloc = new DenseMatrix( new double[][]
                          { {12.,  2. },
                            {15., 13. },
                            {10.,  8. },
                            { 0., 10. },
                            { 6., 13. },
                            { 5.,  8. },
                            {10., 12. },
                            { 4.,  6. },
                            { 5.,  2. },
                            { 1., 10. } } );


  static int N = customerloc.numRows();
```

Then we add the variable defining the facility location, and variables measuring the various distances as described in 4.11:

```
Model M = new Model("FacilityLocation");
// Variable holding the facility location
try
{
  Variable f = M.variable("facility",new NDSet(1,2),Domain.unbounded());
  // Variable defining the euclidian distances to each customer
  Variable d = M.variable("dist", new NDSet(N,1), Domain.greaterThan(0.0));
  // Variable defining the x and y differences to each customer;
  Variable t = M.variable("t",new NDSet(N,2), Domain.unbounded());
```

Here f ("facility") is the (x, y)-location of the facility, d ("dist") is a variable vector of size N measuring the lengths between the facility and the each customers, and t defines an $N \times 2$ matrix of variables measuring the distance in x and y directions for each customer.

Although not strictly necessary, we defines the domains of *dist* and t to be the positive values, while *facility* is unrestricted.

We now wish to model a relation

$$d_i > \sqrt{t_{i,1}^2 + t_{i,2}^2}.$$

This relation is not explicitly written; rather it is formulated using a conic quadratic constraint of size 3:

$$\mathcal{Q}^3 = \{(x_0, x_1, x_2) | x_0^2 > x_1^2 + x_2^2; x_0 > 0\}.$$

This is implemented as a constraint for each customer:

```
M.constraint("dist measure",
             Variable.stack(new Variable[][] {{ d, t }}),
             Domain.inQCone(N,3));
```

The constraints defining the x and y distances are defined as follows:

```
Variable fxy = Variable.repeat(f, N);
M.constraint("xy diff", Expr.add(t, fxy), Domain.equalsTo(customerloc));
```

While it may look complicated, in reality we simply set up vectors holding customer x and y locations (`cxvec` and `cyvec` resp.), two vectors with the x and y facility location (`fxvec` and `fyvec` resp.), and two vectors with the x and y differences (`tx` and `ty` resp.). Finally, we add the equality constraints that define the actual relationship.

We define the objective as minimizing the sum of euclidean distances to each customer:

```
M.objective("total_dist", ObjectiveSense.Minimize, Expr.sum(d));
```

The name "`total_dist`" is optional; it serves no direct purpose in the model, but may be useful when debugging the model.

Finally, we solve the model and print the result:

```
M.solve();
```

```
double[] res = f.level();
System.out.println("Facility location = " + res[0] + "," + res[1]);
```

We find that the optimal location of the facility is $(5.49, 8.14)$ (see fig. 4.3).

4.4.1 Source code: Facility location

```
package com.mosek.fusion.examples;
////
//  Copyright: Copyright (c) MOSEK ApS, Denmark. All rights reserved.
//
//  File:      facility_location.java
```
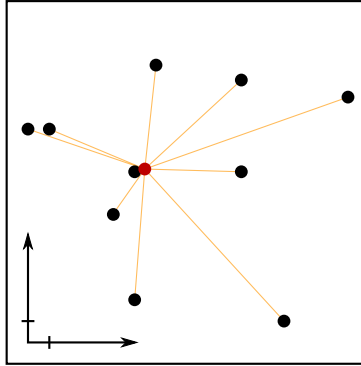
Figure 4.3: Customer locations and optimal placement for facility.

```
//
//  Purpose: Demonstrates a small one-facility location problem.
//
//  Given 10 customers placed in a grid we wish to place a facility
//  somewhere so that the total sum of distances to customers is
//  minimized.
//
//  The problem is formulated as a conic optimization problem as follows.
//  Let f=(fx,fy) be the (unknown) location of the facility, and let
//  c_i=(cx_i,cy_i) be the (known) customer locations; then we wish to
//  minimize
//      sum_i || f - c_i ||
//  where
//      ||.||
//  denotes the euclidian norm.
//  This is formulated as
//
//  minimize    sum(d_i)
//  such that   d_i ^ 2 > tx_i ^ 2 + ty_i ^ 2, for all i
//              tx_i = cx_i - fx, for all i
//              ty_i = cy_i - fy, for all i
//              d_i > 0, for all i
////

import mosek.fusion.*;




public class facility_location
{
// Customer locations
  private static Matrix
    customerloc = new DenseMatrix( new double[][]
                                  { {12.,  2. },
                                    {15., 13. },
                                    {10.,  8. },
```

```
                                      {  0., 10. },
                                      {  6., 13. },
                                      {  5.,  8. },
                                      { 10., 12. },
                                      {  4.,  6. },
                                      {  5.,  2. },
                                      {  1., 10. } } );


    static int N = customerloc.numRows();
  public static void main(String[] args)
    throws SolutionError
  {
    Model M = new Model("FacilityLocation");
    // Variable holding the facility location
    try
    {
      Variable f = M.variable("facility",new NDSet(1,2),Domain.unbounded());
      // Variable defining the euclidian distances to each customer
      Variable d = M.variable("dist", new NDSet(N,1), Domain.greaterThan(0.0));
      // Variable defining the x and y differences to each customer;
      Variable t = M.variable("t",new NDSet(N,2), Domain.unbounded());
      M.constraint("dist measure",
                   Variable.stack(new Variable[][] {{ d, t }}),
                   Domain.inQCone(N,3));

      Variable fxy = Variable.repeat(f, N);
      M.constraint("xy diff", Expr.add(t, fxy), Domain.equalsTo(customerloc));


      M.objective("total_dist", ObjectiveSense.Minimize, Expr.sum(d));

      M.solve();

      double[] res = f.level();
      System.out.println("Facility location = " + res[0] + "," + res[1]);
    }
    finally
    {
      M.dispose();
    }
  }
}
```

4.5   Inner and outer Löwner-John Ellipsoids

Computes the Löwner-John inner and outer ellipsoidal approximations of a polytope.

The inner ellipsoidal approximation to a polytope

$$S = \{x \in \mathbb{R}^n \mid Ax \leq b\}$$

maximizes the volume of the inscribed ellipsoid,

$$\{x \mid x = Cu + d, \ \|u\|_2 \le 1\}.$$

The volume is proportional to $\det(C)^{1/n}$, so the problem can be solved as

$$
\begin{aligned}
\text{maximize} \quad & t \\
\text{subject to} \quad & t \le \det(C)^{1/n} \\
& \|Ca_i\|_2 \le b_i - a_i^T d, \ \ i = 1, \dots, m \\
& C \succeq 0
\end{aligned}
$$

which is equivalent to a mixed conic quadratic and semidefinite programming problem.

The outer ellipsoidal approximation to a polytope given as the convex hull of a set of points

$$S = \text{conv}\{x_1, x_2, \dots, x_m\}$$

minimizes the volume of the enclosing ellipsoid,

$$\{x \mid \|P(x - c)\|_2 \le 1\}$$

The volume is proportional to $\det(P)^{-1/n}$, so the problem can be solved as

$$
\begin{aligned}
\text{minimize} \quad & t \\
\text{subject to} \quad & t \ge \det(P)^{-1/n} \\
& \|Px_i + c\|_2 \le 1, \ \ i = 1, \dots, m \\
& P \succeq 0.
\end{aligned}
$$

4.5.1 Source code: Löwner-John inner and outer ellipsoids

```
package com.mosek.fusion.examples;
/*
   Copyright: Copyright (c) MOSEK ApS, Denmark. All rights reserved.

   File:      lownerjohn_ellipsoid.java

   Purpose:
   Computes the Lowner-John inner and outer ellipsoidal
   approximations of a polytope.


   The inner ellipsoidal approximation to a polytope

      S = { x \in R^n | Ax < b }.

   maximizes the volume of the inscribed ellipsoid,

      { x | x = C*u + d, || u ||_2 <= 1 }.

   The volume is proportional to det(C)^(1/n), so the
   problem can be solved as
```

```
    maximize            t
    subject to          t           <= det(C)^(1/n)
                    || C*ai ||_2 <= bi - ai^T * d,   i=1,...,m
                      C is PSD

which is equivalent to a mixed conic quadratic and semidefinite
programming problem.


The outer ellipsoidal approximation to a polytope given
as the convex hull of a set of points

    S = conv{ x1, x2, ... , xm }

minimizes the volume of the enclosing ellipsoid,

   { x | || P*(x-c) ||_2 <= 1 }

The volume is proportional to det(P)^{-1/n}, so the problem can
be solved as

    minimize            t
    subject to          t           >= det(P)^(-1/n)
                    || P*xi + c ||_2 <= 1,   i=1,...,m
                      P is PSD.

References:
  [1] "Lectures on Modern Optimization", Ben-Tal and Nemirovski, 2000.
*/

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Polygon;
import java.awt.geom.Ellipse2D;
import java.awt.geom.AffineTransform;

import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;


import mosek.fusion.*;

public class lownerjohn_ellipsoid
{
  /**
     Purpose: Models the convex set

        S = { (x, t) \in R^n x R | x >= 0, t <= (x1 * x2 * ... *xn)^(1/n) }.

     as the intersection of rotated quadratic cones and affine hyperplanes,
     see [1, p. 105].  This set can be interpreted as the hypograph of the
     geometric mean of x.

     We illustrate the modeling procedure using the following example.
```

```
    Suppose we have

        t <= (x1 * x2 * x3)^(1/3)

    for some t >= 0, x >= 0. We rewrite it as

        t^4 <= x1 * x2 * x3 * x4,    x4 = t

    which is equivalent to (see [1])

        x11^2 <= 2*x1*x2,    x12^2 <= 2*x3*x4,

        x21^2 <= 2*x11*x12,

        sqrt(8)*x21 = t, x4 = t.

    References:
    [1] "Lectures on Modern Optimization", Ben-Tal and Nemirovski, 2000.
*/
public static void geometric_mean(Model M, Variable x, Variable t)
{
  int n = (int) x.size();
  int l = (int)Math.ceil(Math.log(n)/Math.log(2));
  int m = pow2(l) - n;

  Variable x0;

  if (m == 0)
      x0 = x;
  else
      x0 = Variable.vstack(x, M.variable(m, Domain.greaterThan(0.0)));

  Variable z = x0;

  for (int i = 0; i < l-1; ++i)
  {
    Variable xi = M.variable(pow2(l-i-1), Domain.greaterThan(0.0));
    for (int k = 0; k < pow2(l-i-1); ++k)
      M.constraint(Variable.vstack(z.index(2*k),z.index(2*k+1),xi.index(k)),
                   Domain.inRotatedQCone());
      z = xi;
  }

  Variable t0 = M.variable(1, Domain.greaterThan(0.0));
  M.constraint(Variable.vstack(z, t0), Domain.inRotatedQCone());

  M.constraint(Expr.sub(Expr.mul(Math.pow(2,0.5*l),t),t0), Domain.equalsTo(0.0));

  for (int i = pow2(l-m); i < pow2(l); ++i)
    M.constraint(Expr.sub(x0.index(i), t), Domain.equalsTo(0.0));
}

/**
    Purpose: Models the hypograph of the n-th power of the
    determinant of a positive definite matrix.

    The convex set (a hypograph)
```

```
      C = { (X, t) \in S^n_+ x R |  t <= det(X)^{1/n} },

    can be modeled as the intersection of a semidefinite cone

      [ X, Z; Z^T Diag(Z) ] >= 0

    and a number of rotated quadratic cones and affine hyperplanes,

      t <= (Z11*Z22*...*Znn)^{1/n}  (see geometric_mean).

    References:
    [1] "Lectures on Modern Optimization", Ben-Tal and Nemirovski, 2000.
 */

public static void det_rootn(Model M, Variable X, Variable t)
{
  int n = X.shape.dim(0);

  // Setup variables
  Variable Y = M.variable(Domain.inPSDCone(2*n));

  // Setup Y = [X, Z; Z^T diag(Z)]
  Variable Y11 = Y.slice(new int[] {0, 0}, new int[] {n, n});
  Variable Y21 = Y.slice(new int[] {n, 0}, new int[] {2*n, n});
  Variable Y22 = Y.slice(new int[] {n, n}, new int[] {2*n, 2*n});

  double[] ones= new double[n];
  for (int i = 0; i < n; ++i)
    ones[i] = 1.0;

  Matrix S = Matrix.sparse(n, n, irange(0,n), irange(0,n), ones);
  M.constraint( Expr.sub(Expr.mulElm(S,Y21), Y22), Domain.equalsTo(0.0));
  M.constraint( Expr.sub(X, Y11), Domain.equalsTo(0.0) );

  // t^n <= (Z11*Z22*...*Znn)
  Variable[] tmpv = new Variable[n];
  for (int i = 0; i < n; ++i) tmpv[i] = Y22.index(i,i);
  Variable z = Variable.reshape(Variable.vstack(tmpv), n);
  geometric_mean(M, z, t);
}

public static int[] irange(int first, int last)
{
  int[] r = new int[last-first];
  for (int i = 0; i < last-first; ++i)
    r[i] = i+first;
  return r;
}

public static int pow2(int n)
{
  return (int) (1 << n);
}

public static double[] lownerjohn_inner(double[][] A, double[] b)
  throws SolutionError
{
  Model M = new Model("lownerjohn_inner");
```

```java
    try
    {
      int m = A.length;
      int n = A[0].length;

      // Setup variables
      Variable t = M.variable("t", 1, Domain.greaterThan(0.0));
      Variable C = M.variable("C", new NDSet(n,n), Domain.unbounded());
      Variable d = M.variable("d", n, Domain.unbounded());

      // (bi - ai^T*d, C*ai) \in Q
      for (int i = 0; i < m; ++i)
        M.constraint( "qc" + i, Expr.vstack(Expr.sub(b[i],Expr.dot(A[i],d)), Expr.mul(C,A[i])), Domain.inQCone()
);

      // t <= det(C)^{1/n}
      //model_utils.det_rootn(M, C, t);
      det_rootn(M, C, t);

      // Objective: Maximize t
      M.objective(ObjectiveSense.Maximize, t);
      M.solve();

      double[] Clvl = C.level();
      double[] dlvl = d.level();

      double[] Cd = new double[Clvl.length + dlvl.length];

      for(int j = 0; j < n; ++j)
      {
        for (int i = 0; i < n; ++i)
        {
          System.out.println("i,j = " + i + "," + j);
          Cd[n*i+j] = Clvl[n*j+i];
        }
        Cd[n*n+j] = dlvl[j];
      }

      System.out.print("C  = [ " + Clvl[0]); for (int i = 1; i < Clvl.length; ++i) System.out.print("," +
Clvl[i]); System.out.println(" ]");
      System.out.print("d  = [ " + dlvl[0]); for (int i = 1; i < dlvl.length; ++i) System.out.print("," +
dlvl[i]); System.out.println(" ]");
      System.out.print("Cd = [ " + Cd[0]);   for (int i = 1; i < Cd.length; ++i) System.out.print("," + Cd[i]);
System.out.println(" ]");

      return Cd;
    }
    finally
    {
      M.dispose();
    }
  }


  public static double[] lownerjohn_outer(double[][] x)
    throws SolutionError
  {
    Model M = new Model("lownerjohn_outer");
```

```java
    try
    {
      int m = x.length;
      int n = x[0].length;

      // Setup variables
      Variable t = M.variable("t", 1, Domain.greaterThan(0.0));
      Variable P = M.variable("P", new NDSet(n,n), Domain.unbounded());
      Variable c = M.variable("c", n, Domain.unbounded());

      // (1, P(*xi+c)) \in Q
      for (int i = 0; i < m; ++i)
        M.constraint( "qc" + i, Expr.vstack(Expr.ones(1), Expr.sub(Expr.mul(P,x[i]), c)), Domain.inQCone()
);

      // t <= det(P)^{1/n}
      //model_utils.det_rootn(M, P, t);
      det_rootn(M, P, t);

      // Objective: Maximize t
      M.objective(ObjectiveSense.Maximize, t);
      M.solve();

      double[] Plvl = P.level();
      double[] clvl = c.level();

      double[] Pc = new double[Plvl.length + clvl.length];


      for(int j = 0; j < n; ++j)
      {
        for (int i = 0; i < n; ++i)
          Pc[n*i+j] = Plvl[n*j+i];
        Pc[n*n+j] = clvl[j];
      }
      return Pc;
    }
    finally
    {
      M.dispose();
    }
  }


  public static void main(String[] argv)
    throws SolutionError
  {
    final double[][] p = { {0.,0.},{1.,3.}, {5.,4.}, {7.,1.}, {3.,-2.} };
    double[][] A = new double[p.length][];
    double[]   b = new double[p.length];
    int n = p.length;
    for (int i = 0; i < n; ++i)
    {
      A[i] = new double[] { -p[i][1]+p[(i-1+n)%n][1], p[i][0]-p[(i-1+n)%n][0] };
      b[i] = A[i][0]*p[i][0]+A[i][1]*p[i][1];
    }

    double[] Cd = lownerjohn_inner(A, b);
```

```
    double[] Pc = lownerjohn_outer(p);
  }
}
```

4.6 Nearest correlation

In the nearest correlation problem we are given a symmetric matrix A and we wish to find the closest correlation matrix, measured by the Frobineus norm.

In other words, the nearest correlation matrix is given by

$$X^\star = \arg\min_{x \in S} \|A - X\|_F$$

where

$$S = \{X \in \mathbf{R}^{n \times n} \mid X \succeq 0, \; \mathbf{diag}(X) = e\}.$$

To exploit symmetry of $A - X$ we define a mapping from a symmetric matrix to vector containing the (scaled) lower triangular part of the matrix,

$$\mathbf{vec}(U) = (U_{11}, \sqrt{2}U_{21}, \ldots, \sqrt{2}U_{n1}, U_{22}, \sqrt{2}U_{32}, \ldots, \sqrt{2}U_{n2}, \ldots, U_{nn}),$$

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

We then get an optimization problem with both conic quadratic and semidefinite constraints,

$$
\begin{array}{ll}
\text{minimize} & t \\
\text{subject to} & \|z\|_2 \leq t \\
& \mathrm{vec}(A - X) = z \\
& \mathrm{diag}(X) = e \\
& X \succeq 0, \\
\text{where} & \\
& \mathrm{vec}: M^{n \times n} \to R^{n(n+1)/2} \\
& \mathrm{vec}(M)_k = M_{ij} \text{ for } k = i(i+1)/2 + j, \; i = j \\
& \mathrm{vec}(M)_k = \sqrt{2}M_{ij} \text{ for } k = i(i+1)/2, \; i < j.
\end{array}
$$

4.6.1 Source code: Nearest correlation

```java
package com.mosek.fusion.examples;
/*
   Copyright: Copyright (c) MOSEK ApS, Denmark. All rights reserved.

   File:      nearestcorr.java

   Purpose:
   Solves the nearest correlation matrix problem
```

```
     minimize   || A - X ||_F   s.t.  diag(X) = e, X is PSD

  as the equivalent conic program

     minimize     t

     subject to   (t, vec(A-X)) in Q
                  diag(X) = e
                  X >= 0
     where
                  vec : M(n x n) -> R^(n*(n+1)/2)
                  vec(M)_k = M_ij          for k = i * (i+1) / 2 + j, and i == j
                           | sqrt(2) * M_ij for k = i * (i+1) / 2, i < j
*/
import mosek.fusion.*;

public class nearestcorr
{
  /** Assuming that e is an NxN expression, return the lower triangular part as a vector.
  */
  public static Expression vec(Expression e)
  {
    int N     = e.getShape().dim(0);
    int[] msubi = new int[N*(N+1)/2],
          msubj = new int[N*(N+1)/2];
    double[] mcof = new double[N*(N+1)/2];

    {
      for (int i = 0, k = 0; i < N; ++i)
        for (int j = 0; j < i+1; ++j, ++k)
        {
          msubi[k] = k;
          msubj[k] = i*N+j;
          if (i == j) mcof[k] = 1.0;
          else        mcof[k] = Math.sqrt(2);

        }
    }

    Matrix S = Matrix.sparse(N * (N+1) / 2,N * N,msubi,msubj,mcof);

    return Expr.mul(S,Expr.reshape( e, N * N ));
  }
  private static String arrtostr(double[] a)
  {
    StringBuilder b = new StringBuilder("[");
    java.util.Formatter f = new java.util.Formatter(b, java.util.Locale.US);
    if (a.length > 0) f.format(", %.3e", a[0]);
    for (int i = 1; i < a.length; ++i) f.format(", %.3e", a[i]);
    b.append("]");
    return b.toString();
  }
  public static void main(String[] argv)
    throws SolutionError
  {
    int N = 5;
    double[][] A = new double[][]
        { { 0.0,  0.5,  -0.1,  -0.2,   0.5},
```

```
                   { 0.5,  1.25, -0.05, -0.1,   0.25},
                   {-0.1, -0.05,  0.51,  0.02, -0.05},
                   {-0.2, -0.1,   0.02,  0.54, -0.1},
                   { 0.5,  0.25, -0.05, -0.1,   1.25} };

   // Create a model with the name 'NearestCorrelation
   Model M = new Model("NearestCorrelation");
   try
   {
     // Setting up the variables
     Variable X = M.variable("X", Domain.inPSDCone(N));
     Variable t = M.variable("t", 1, Domain.unbounded());

     // (t, vec (A-X)) \in Q
     M.constraint( Expr.vstack(t, vec(Expr.sub(new DenseMatrix(A),X))), Domain.inQCone() );

     // diag(X) = e
     M.constraint(X.diag(), Domain.equalsTo(1.0));

     // Objective: Minimize t
     M.objective(ObjectiveSense.Minimize, t);

     // Solve the problem
     M.solve();

     // Get the solution values
     System.out.println("X = " + arrtostr(X.level()));

     System.out.println("t = " + arrtostr(t.level()));
   }
   finally
   {
     M.dispose();
   }
 }
}
```

Chapter 5

Fusion Class reference

5.1 `fusion.BaseSet`

Base class for 1-dimensional sets.

Base class
> `Set`

Known direct descendants
> `IntSet`, `StringSet`

Constructors

> `BaseSet(long size)`
>> Constructor for simple one-dimensional sets.

Methods

> `BaseSet.dim`
>> Return the size of the given dimension.

Inherited methods

> `Set.compare(Set other)`
>> Compare two sets and return true if they have the same shape and size.
>
> `Set.getSize()`
>> Total number of elements in the set.
>
> `Set.getname(int[] key)`
>> Return a string representing the index.

Set.getname(long key)

> Return a string representing the index.

Set.idxtokey(long idx)

> Convert a linear index to a N-dimensional key.

Set.indexToString(long index)

Set.realnd()

> Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

Set.slice(int first,int last)

> Create a set object representing a slice of this set.

Set.slice(int[] first,int[] last)

> Create a set object representing a slice of this set.

Set.stride(int i)

> Return the stride size in the given dimension.

Set.toString()

> Return a string representation of the set.

5.1.1 BaseSet.BaseSet()

Member of BaseSet.

BaseSet.BaseSet(long size)

5.1.1.1 BaseSet.BaseSet(size)

Constructor for simple one-dimensional sets.

size : long

> Number of items in the set.

5.1.2 BaseSet.dim()

Member of BaseSet.

BaseSet.dim(int i)

5.1.2.1 BaseSet.dim(i)

Return the size of the given dimension.

i : int

> Dimension index.

returns : `int`

> The size of the requested dimension.

5.2   `fusion.BoundInterfaceConstraint`

Interface to either the upper bound or the lower bound of a ranged constraint.

This class is never explicitly instantiated; is is created by a `RangedConstraint` to allow accessing a bound value and the dual variable value corresponding to the relevant bound as a separate object. The constraint

$$b_l \leq a^t x \leq b_u$$

has two bounds and two dual variables; these are not immediately available through the `RangedConstraint` object, but can be accessed through a `BoundInterfaceConstraint`.

Base class

> `SliceConstraint`

Inherited methods

> `Constraint.add(Expression expr)`
>> Add an expression to the constraint expression.
>
> `Constraint.add(Variable v)`
>> Add an expression to the constraint expression.
>
> `Constraint.dual()`
>> Get the dual solution value of the variable.
>
> `Constraint.dual(int[] firstidx,int[] lastidx)`
>> Return a slice of the dual solution.
>
> `Constraint.dual(int firstidx,int lastidx)`
>> Return a slice of the dual solution.
>
> `Constraint.get_model()`
>> Get the original model object.
>
> `Constraint.get_nd()`
>> Get the number of dimensions of the constraint.
>
> `Constraint.index(int[] idx)`
>> Get a single element from a one-dimensional constraint.
>
> `Constraint.index(int idx)`
>> Get a single element from a one-dimensional constraint.
>
> `Constraint.level(int index)`
>> Return a single value of the primal solution.

`Constraint.level(int firstidx,int lastidx)`
>   Return a slice of the primal solution.

`Constraint.level()`
>   Get the primal solution value of the variable.

`Constraint.level(int[] firstidx,int[] lastidx)`
>   Return a slice of the primal solution.

`SliceConstraint.size()`
>   Get the total number of elements in the constraint.

`SliceConstraint.slice(int firstidx,int lastidx)`
>   Get a slice of the constraint.

`SliceConstraint.slice(int[] firstidx,int[] lastidx)`
>   Get a multi-dimensional slice of the constraint.

`Constraint.toString()`

5.3   `fusion.BoundInterfaceVariable`

Interface to either the upper bound or the lower bound of a ranged variable.

This class is never explicitly instantiated; is is created by a `RangedVariable` to allow accessing a bound value and the dual variable value corresponding to the relevant bound as a separate object. The variable

$$b_l \leq x \leq b_u$$

has two bounds and two dual variables; these are not immediately available through the `RangedVariable` object, but can be accessed through a `BoundInterfaceVariable`.

Base class

>   `SliceVariable`

Inherited methods

`Variable.antidiag(int index)`
>   Return an anti-diagonal of a square matrix.

`Variable.antidiag()`
>   Return the anti-diagonal of a square matrix

`Variable.asExpr()`
>   Create an expression corresponding to the variable object.

`Variable.compress()`
>   Reshape a variable object by removing all dimensions of size 1.

`Variable.diag()`
    Return the diagonal of a square matrix

`Variable.diag(int index)`
    Return a diagonal of a square matrix.

`Variable.dual(int[] firstidx,int[] lastidx)`
    Return a slice of the dual solution.

`Variable.dual(int firstidx,int lastidx)`
    Return a slice of the dual solution.

`Variable.dual()`
    Get the dual solution value of the variable.

`Variable.index(int[] idx)`
    Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1)`
    Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1,int i2)`
    Get a single element from a one-dimensional variable.

`Variable.index(int i0)`
    Get a single element from a one-dimensional variable.

`Variable.index_flat(long i)`
    Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`
    Return a single value of the primal solution.

`Variable.level()`
    Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`
    Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`
    Return a slice of the primal solution.

`Variable.pick(int[] idxs)`
    Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`
    Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`
    Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`
    Get the number of elements in the variable.

`SliceVariable.slice(int[] firstidx,int[] lastidx)`
    Get a multi-dimensional slice of the variable.

`SliceVariable.slice(int firstidx,int lastidx)`
    Create a variable object representing a slice of the variable.

`Variable.toString()`
    Create a string-representation of the variable.

`Variable.transpose()`
    Transpose a vector or matrix variable

5.4   `fusion.CompoundConstraint`

Stacking of constraints.

A `CompoundConstraint` represents a stack og other variable objects and can be used as a 1-dimensional variable. The class is never explicitly instantiated, but is created using `Constraint.stack`.

As this class is derived from `Variable`, it may be used as a normal variable when creating expressions.

Base class
    `Constraint`

Constructors
    `CompoundConstraint(Constraint[] c)`

Methods
    `CompoundConstraint.slice`

Inherited methods

`Constraint.add(Expression expr)`
    Add an expression to the constraint expression.

`Constraint.add(Variable v)`
    Add an expression to the constraint expression.

`Constraint.dual()`
    Get the dual solution value of the variable.

`Constraint.dual(int[] firstidx,int[] lastidx)`
    Return a slice of the dual solution.

`Constraint.dual(int firstidx,int lastidx)`
    Return a slice of the dual solution.

`Constraint.get_model()`
    Get the original model object.

Constraint.get_nd()
> Get the number of dimensions of the constraint.

Constraint.index(int[] idx)
> Get a single element from a one-dimensional constraint.

Constraint.index(int idx)
> Get a single element from a one-dimensional constraint.

Constraint.level(int index)
> Return a single value of the primal solution.

Constraint.level(int firstidx,int lastidx)
> Return a slice of the primal solution.

Constraint.level()
> Get the primal solution value of the variable.

Constraint.level(int[] firstidx,int[] lastidx)
> Return a slice of the primal solution.

Constraint.size()
> Get the total number of elements in the constraint.

Constraint.toString()

5.4.1 CompoundConstraint.CompoundConstraint()

Member of CompoundConstraint.

CompoundConstraint.CompoundConstraint(Constraint[] c)

5.4.1.1 CompoundConstraint.CompoundConstraint(c)

Constructor. Given a list of constraints, create a 1D alias variable.

Please note that the constraints will be unfolded into a 1D vector, so be careful!

c : Constraint[]

Throws exceptions :

- ModelError

5.4.2 CompoundConstraint.slice()

Member of CompoundConstraint.

CompoundConstraint.slice(int first,int last)
CompoundConstraint.slice(int[] first,int[] last)

5.4.2.1   CompoundConstraint.slice(first,last)

Get a slice of the constraint.

Assuming that the shape of the constraint is one-dimensional, this function creates a constraint slice of (last-first+1) elements, which acts as an interface to a sub-vector of this constraint.

`first :  int`
>   Index of the first element in the slice.

`last :  int`
>   Index if the last element in the slice.

returns : `Constraint`
>   A new constraint object representing a slice of this object.

5.4.2.2   CompoundConstraint.slice(first,last)

Get a multi-dimensional slice of the constraint.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the constraint.

`first :  int[]`
>   Array of start elements in the slice.

`last :  int[]`
>   Array of end element in the slice.

returns : `Constraint`
>   A new constraint object representing a slice of this object.

5.5   `fusion.CompoundVariable`

A stack of several other variables.

A `CompoundVariable` represents a stack og other variable objects and can be used as a 1-dimensional variable. The class is never explicitly instantiated, but is created using `Variable.stack`.

Base class
>   `Variable`

Methods

>   `CompoundVariable.asExpr`
>>   Create an expression corresponding to the variable object.

CompoundVariable.slice

Inherited methods

Variable.antidiag(int index)

Return an anti-diagonal of a square matrix.

Variable.antidiag()

Return the anti-diagonal of a square matrix

Variable.compress()

Reshape a variable object by removing all dimensions of size 1.

Variable.diag()

Return the diagonal of a square matrix

Variable.diag(int index)

Return a diagonal of a square matrix.

Variable.dual(int[] firstidx,int[] lastidx)

Return a slice of the dual solution.

Variable.dual(int firstidx,int lastidx)

Return a slice of the dual solution.

Variable.dual()

Get the dual solution value of the variable.

Variable.index(int[] idx)

Get a single element from a one-dimensional variable.

Variable.index(int i0,int i1)

Get a single element from a one-dimensional variable.

Variable.index(int i0,int i1,int i2)

Get a single element from a one-dimensional variable.

Variable.index(int i0)

Get a single element from a one-dimensional variable.

Variable.index_flat(long i)

Index into the variable as if it was a one-dimensional object.

Variable.level(int index)

Return a single value of the primal solution.

Variable.level()

Get the primal solution value of the variable.

Variable.level(int[] firstidx,int[] lastidx)

Return a slice of the primal solution.

Variable.level(int firstidx,int lastidx)

Return a slice of the primal solution.

`Variable.pick(int[] idxs)`

    Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`

    Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`

    Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`

    Get the number of elements in the variable.

`Variable.toString()`

    Create a string-representation of the variable.

`Variable.transpose()`

    Transpose a vector or matrix variable

5.5.1 CompoundVariable.slice()

Member of `CompoundVariable`.

```
CompoundVariable.slice(int first,int last)
CompoundVariable.slice(int[] first,int[] last)
```

5.5.1.1 CompoundVariable.slice(first,last)

Get a slice of the variable.

Assuming that the shape of the variable is one-dimensional, this function creates a variable slice of (last-first+1) elements, which acts as an interface to a sub-vector of this variable.

`first : int`

    Index of the first element in the slice.

`last : int`

    Index if the last element plus one in the slice.

returns : `Variable`

    A new variable object representing a slice of this object.

5.5.1.2 CompoundVariable.slice(first,last)

Get a multi-dimensional slice of the variable.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the variable.

`first : int[]`

> Array of start elements in the slice.

`last : int[]`

> Array of end elements plus one in the slice.

returns : `Variable`

> A new variable object representing a slice of this object.

5.5.2 `CompoundVariable.asExpr()`

Member of `CompoundVariable`.

`CompoundVariable.asExpr()`

5.5.2.1 **CompoundVariable.asExpr()**

Create an `Expression` object corresponding to $I \cdot V$, where I is the identity matrix and V is this variable.

returns : `Expression`

> An Expression object representing the product I*V, where I is the identity matrix and V is this variable.

5.6 `fusion.ConicConstraint`

This class represents a conic constraint of the form

$$Ax - b \in \mathbb{C}$$

where $\mathbb{C}$ is either a quadratic cone or a rotated quadratic cone. Then class is never explicitly instantiated, but is created using `Model.constraint` by specifying a conic domain.

Note that a conic constraint in Fusion is always "dense" in the sense that all member constraints are created in the underlying optimization problem immediately.

Base class

> `ModelConstraint`

Inherited methods

> `Constraint.add(Expression expr)`
>
> > Add an expression to the constraint expression.

Constraint.add(Variable v)
>   Add an expression to the constraint expression.

Constraint.dual()
>   Get the dual solution value of the variable.

Constraint.dual(int[] firstidx,int[] lastidx)
>   Return a slice of the dual solution.

Constraint.dual(int firstidx,int lastidx)
>   Return a slice of the dual solution.

Constraint.get_model()
>   Get the original model object.

Constraint.get_nd()
>   Get the number of dimensions of the constraint.

Constraint.index(int[] idx)
>   Get a single element from a one-dimensional constraint.

Constraint.index(int idx)
>   Get a single element from a one-dimensional constraint.

Constraint.level(int index)
>   Return a single value of the primal solution.

Constraint.level(int firstidx,int lastidx)
>   Return a slice of the primal solution.

Constraint.level()
>   Get the primal solution value of the variable.

Constraint.level(int[] firstidx,int[] lastidx)
>   Return a slice of the primal solution.

Constraint.size()
>   Get the total number of elements in the constraint.

ModelConstraint.slice(int first,int last)
>   Get a slice of the constraint.

ModelConstraint.slice(int[] first,int[] last)
>   Get a multi-dimensional slice of the constraint.

ModelConstraint.toString()

5.7 fusion.ConicVariable

This class represents a conic variable of the form

$$Ax - b \in \mathbb{C}$$

where $\mathbb{C}$ is either a quadratic cone or a rotated quadratic cone. Then class is never explicitly instantiated, but is created using `Model.variable` by specifying a conic domain.

Note that a conic variable in Fusion is always "dense" in the sense that all member variables are created in the underlying optimization problem immediately.

Base class

> `ModelVariable`

Known direct descendants

> `IntegerConicVariable`

Inherited methods

> `Variable.antidiag(int index)`
>> Return an anti-diagonal of a square matrix.

> `Variable.antidiag()`
>> Return the anti-diagonal of a square matrix

> `Variable.asExpr()`
>> Create an expression corresponding to the variable object.

> `Variable.compress()`
>> Reshape a variable object by removing all dimensions of size 1.

> `Variable.diag()`
>> Return the diagonal of a square matrix

> `Variable.diag(int index)`
>> Return a diagonal of a square matrix.

> `Variable.dual(int[] firstidx,int[] lastidx)`
>> Return a slice of the dual solution.

> `Variable.dual(int firstidx,int lastidx)`
>> Return a slice of the dual solution.

> `Variable.dual()`
>> Get the dual solution value of the variable.

> `Variable.index(int[] idx)`
>> Get a single element from a one-dimensional variable.

> `Variable.index(int i0,int i1)`
>> Get a single element from a one-dimensional variable.

> `Variable.index(int i0,int i1,int i2)`
>> Get a single element from a one-dimensional variable.

> `Variable.index(int i0)`
>> Get a single element from a one-dimensional variable.

`Variable.index_flat(long i)`

    Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`

    Return a single value of the primal solution.

`Variable.level()`

    Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`

    Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`

    Return a slice of the primal solution.

`Variable.pick(int[] idxs)`

    Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`

    Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`

    Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`

    Get the number of elements in the variable.

`ModelVariable.slice(int[] first,int[] last)`

    Get a multi-dimensional slice of the variable.

`ModelVariable.slice(int first,int last)`

    Create a variable object representing a slice of the variable.

`ModelVariable.toString()`

    Create a string-representation of the variable.

`Variable.transpose()`

    Transpose a vector or matrix variable

5.8   fusion.Constraint

An abstract constraint object. This is the base class for all constraint types in Fusion.

The `Constraint` object can be an interface to the normal model constraint, e.g. `LinearConstraint` and `ConicConstraint`, to slices of other constraints or to concatenations of other constraints.

Primal and dual solution values can be accessed through the `Constraint` object.

  Known direct descendants

    `ModelConstraint`, `CompoundConstraint`, `SliceConstraint`

Constructors

> `Constraint(`Model` model,`Set` shape)`
>> Initialize a constraint belonging to a specified model.

Methods

> `Constraint.add`
>
> `Constraint.dual`
>
> `Constraint.get_model`
>> Get the original model object.
>
> `Constraint.get_nd`
>> Get the number of dimensions of the constraint.
>
> `Constraint.index`
>
> `Constraint.level`
>
> `Constraint.size`
>> Get the total number of elements in the constraint.
>
> `Constraint.slice`
>
> `Constraint.toString`

Static Methods

> `Constraint.stack`

5.8.1   `Constraint.Constraint()`

Member of `Constraint`.

`Constraint.Constraint(`Model` model,`Set` shape)`

5.8.1.1   **Constraint.Constraint(model,shape)**

Initialize a constraint belonging to a specified model.

`model` :   `Model`
> The `Model` object to which the constraint belongs.

`shape` :   `Set`
> A `Set` object defining the shape of the constraint.

5.8.2   `Constraint.index()`

Member of `Constraint`.

`Constraint.index(int idx)`
`Constraint.index(int[] idx)`

5.8.2.1   Constraint.index(idx)

Get a single element from a one-dimensional constraint.

`idx : int`

>   The element index.

`returns : Constraint`

>   A new slice containing a single element.

5.8.2.2   Constraint.index(idx)

Get a single element from a one-dimensional constraint.

`idx : int[]`

>   Array of integers entry in each dimension.

`returns : Constraint`

>   A new slice containing a single element.

5.8.3   Constraint.slice()

Member of `Constraint`.

```
abstract Constraint.slice(int first,int last)
abstract Constraint.slice(int[] first,int[] last)
```

5.8.3.1   abstract Constraint.slice(first,last)

Get a slice of the constraint.

Assuming that the shape of the constraint is one-dimensional, this function creates a constraint slice of (last-first+1) elements, which acts as an interface to a sub-vector of this constraint.

`first : int`

>   Index of the first element in the slice.

`last : int`

>   Index if the last element in the slice.

`returns : Constraint`

>   A new constraint object representing a slice of this object.

5.8.3.2   abstract Constraint.slice(first,last)

Get a multi-dimensional slice of the constraint.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the constraint.

`first :  int[]`

> Array of start elements in the slice.

`last :  int[]`

> Array of end element in the slice.

returns : `Constraint`

> A new constraint object representing a slice of this object.

5.8.4   Constraint.get_nd()

Member of `Constraint`.

`Constraint.get_nd()`

5.8.4.1   Constraint.get_nd()

Get the number of dimensions of the constraint.

returns : `int`

> The number of dimensions in the constraint.

5.8.5   Constraint.level()

Member of `Constraint`.

```
Constraint.level(int firstidx,int lastidx)
Constraint.level(int[] firstidx,int[] lastidx)
Constraint.level(int index)
Constraint.level()
```

5.8.5.1   Constraint.level(firstidx,lastidx)

Return a slice of the primal solution from a one-dimensional variable. If the variable is not one-dimensional, an exception will be thrown. The solution values are taken from the default solution defined in the `Model`.

`firstidx :  int`

> Index of the first element in the range.

`lastidx :  int`

>   Index of the last element (inclusive) in the range.

Throws exceptions :

>   - `SolutionError`

returns : `double[]`

>   An array of solution values.

5.8.5.2   Constraint.level(firstidx,lastidx)

Return a slice of the primal solution from a variable. If the length of the index arrays does not match
the number of dimensions in the variable, an exception will be thrown. The solution values are taken
from the default solution defined in the `Model`.

`firstidx :  int[]`

>   Array of indexes of the first element in each dimension.

`lastidx :  int[]`

>   Array of indexes of the last element (inclusive) in each dimension.

Throws exceptions :

>   - `SolutionError`

returns : `double[]`

>   An array of solution values. When the selected slice is multi-dimensional, this corresponds to
>   the flattened slice of solution values.

5.8.5.3   Constraint.level(index)

Return a single solution value from a 1-dimensional variable. This corresponds to getting a 1-
dimensionals slice of size 1.

See `Variable.level` for details.

`index :  int`

>   Index of the element whose solution value to return.

Throws exceptions :

>   - `SolutionError`

returns : `double`

>   The solution value as a double.

5.8.5.4 Constraint.level()

Get the primal solution value of the variable.

Throws exceptions :

- SolutionError

returns : `double[]`

> An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

5.8.6 Constraint.add()

Member of Constraint.

```
Constraint.add(Expression expr)
Constraint.add(Variable v)
```

5.8.6.1 Constraint.add(expr)

Add an expression to the constraint expression.

expr : Expression

returns : Constraint

> The constraint itself.

5.8.6.2 Constraint.add(v)

Add an expression to the constraint expression.

v : Variable

returns : Constraint

> The constraint itself.

5.8.7 Constraint.get_model()

Member of Constraint.

```
Constraint.get_model()
```

5.8.7.1 Constraint.get_model()

Get the original model object.

 returns : Model
>   The model to which the constraint belongs.

5.8.8   Constraint.toString()

Member of Constraint.
 Constraint.toString()

5.8.8.1 Constraint.toString()

 returns : String

5.8.9   Constraint.dual()

Member of Constraint.
 Constraint.dual()
 Constraint.dual(int firstidx,int lastidx)
 Constraint.dual(int[] firstidx,int[] lastidx)

5.8.9.1 Constraint.dual()

Get the dual solution value of the variable.

 Throws exceptions :

> - SolutionError

 returns : double[]
>   An array of values corresponding to the dual solution values of the constraint.

5.8.9.2 Constraint.dual(firstidx,lastidx)

Return a slice of the dual solution from a one-dimensional constraint. If the constraint is not one-dimensional, an exception will be thrown. The solution values are taken from the default solution defined in the Model.

 firstidx :   int
>   Index of the first element in the range.

```
lastidx :  int
```
>   Index of the last element (inclusive) in the range.

Throws exceptions :

> - SolutionError

returns : `double[]`

>   An array of solution values.

5.8.9.3   Constraint.dual(firstidx,lastidx)

Return a slice of the dual solution from a constraint. If the length of the index arrays does not match the number of dimensions in the constraint, an exception will be thrown. The solution values are taken from the default solution defined in the `Model`.

```
firstidx :  int[]
```
>   Array of indexes of the first element in each dimension.

```
lastidx :  int[]
```
>   Array of indexes of the last element (inclusive) in each dimension.

Throws exceptions :

> - SolutionError

returns : `double[]`

>   An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

5.8.10   Constraint.stack()

Member of `Constraint`.
```
static Constraint.stack(Constraint v1,Constraint v2)
static Constraint.stack(Constraint v1,Constraint v2,Constraint v3)
static Constraint.stack(Constraint[] clist)
```

5.8.10.1   static Constraint.stack(v1,v2)

Stack two constraints.

Create a new constraint representing the concatenation of the given constraints.

```
v1 :  Constraint
```
>   The first constraint in the stack.

v2 :   `Constraint`

    The second constraint in the stack.

Throws exceptions :

        • `ModelError`

returns : `Constraint`

    An object representing the concatenation of the constraints.

5.8.10.2   static Constraint.stack(v1,v2,v3)

Stack three constraints.

Create a new constraint representing the concatenation of the given constraints.

v1 :   `Constraint`

    The first constraint in the stack.

v2 :   `Constraint`

    The second constraint in the stack.

v3 :   `Constraint`

    The second constraint in the stack.

Throws exceptions :

        • `ModelError`

returns : `Constraint`

    An object representing the concatenation of the constraints.

5.8.10.3   static Constraint.stack(clist)

Create a stacked constraint.

Create a new constraint representing the concatenation of the given constraints.

clist :   `Constraint`[]

    The constraints in the stack.

Throws exceptions :

        • `ModelError`

returns : `Constraint`

    An object representing the concatenation of the constraints.

5.8.11 Constraint.size()

Member of Constraint.

Constraint.size()

5.8.11.1 Constraint.size()

Get the total number of elements in the constraint.

returns : long
> The total numbe of elements in the constraint.

5.9 fusion.DenseMatrix

A dense $n{\times}m$ matrix.

Base class
> Matrix

Constructors

> DenseMatrix(int dimi_,int dimj_,double value_)
>> Construct a dense matrix with the given size and fill it with the given value.

> DenseMatrix(double[][] d)
>> Construct a dense matrix from data.

> DenseMatrix(Matrix m)
>> Construct a dense matrix from another Matrix object.

> DenseMatrix(int dimi,int dimj,double[] cof)
>> Construct a dense matrix from a one-dimensional array.

Methods

> DenseMatrix.get

> DenseMatrix.getDataAsArray
>> Return the data a dense array of values.

> DenseMatrix.getDataAsTriplets
>> Return the matrix data in triplet format.

> DenseMatrix.isSparse
>> Returns true if the matrix is sparse.

> DenseMatrix.numNonzeros
>> Number of non-zero elements in the matrix.

DenseMatrix.toString
> Get a string representation of the matrix.

DenseMatrix.transpose
> Tranpose the matrix.

Inherited methods

Matrix.numColumns()
> Returns the number columns in the matrix.

Matrix.numRows()
> Returns the number rows in the matrix.

5.9.1   DenseMatrix.DenseMatrix()

Member of DenseMatrix.
```
DenseMatrix.DenseMatrix(int dimi_,int dimj_,double value_)
DenseMatrix.DenseMatrix(double[][] d)
DenseMatrix.DenseMatrix(Matrix m)
DenseMatrix.DenseMatrix(int dimi,int dimj,double[] cof)
```

5.9.1.1   DenseMatrix.DenseMatrix(dimi_,dimj_,value_)

Construct a dense matrix with the given size and fill it with the given value.

dimi_ :   int

dimj_ :   int

value_ :   double

5.9.1.2   DenseMatrix.DenseMatrix(d)

Construct a dense matrix from data.

d :   double[][]
> Two-dimensional matrix of values.

5.9.1.3   DenseMatrix.DenseMatrix(m)

Construct a dense matrix from another Matrix object.

m :   Matrix
> A Matrix object.

5.9.1.4 **DenseMatrix.DenseMatrix(dimi,dimj,cof)**

Construct a dense matrix from a one-dimensional array.

`dimi : int`

> Height of the matrix.

`dimj : int`

> Width of the matrix.

`cof : double[]`

> Matrix coefficients as a one-dimensional array whose must be at least `dimi*dimj`. Elements are ordered by rows, i.e. elements 1..`dimj` is the first row, `dimj` + 1...2`dimj` is the second row etc.

5.9.2 `DenseMatrix.get()`

Member of `DenseMatrix`.

`DenseMatrix.get(int i,int j)`

5.9.2.1 **DenseMatrix.get(i,j)**

`i : int`

`j : int`

`returns : double`

5.9.3 `DenseMatrix.transpose()`

Member of `DenseMatrix`.

`DenseMatrix.transpose()`

5.9.3.1 **DenseMatrix.transpose()**

Tranpose the matrix.

`returns : Matrix`

5.9.4 `DenseMatrix.numNonzeros()`

Member of `DenseMatrix`.

`DenseMatrix.numNonzeros()`

5.9.4.1   DenseMatrix.numNonzeros()

Number of non-zero elements in the matrix.

 returns : `long`
     Number of non-zeros.

5.9.5   DenseMatrix.isSparse()

Member of DenseMatrix.
 DenseMatrix.isSparse()

5.9.5.1   DenseMatrix.isSparse()

Returns true if the matrix is sparse.

 returns : `boolean`

5.9.6   DenseMatrix.toString()

Member of DenseMatrix.
 DenseMatrix.toString()

5.9.6.1   DenseMatrix.toString()

Get a string representation of the matrix.

 returns : `String`
     A string representation of the matrix.

5.9.7   DenseMatrix.getDataAsTriplets()

Member of DenseMatrix.
 DenseMatrix.getDataAsTriplets(int[] subi,int[] subj,double[] cof)

5.9.7.1   DenseMatrix.getDataAsTriplets(subi,subj,cof)

Return the matrix data in triplet format. Data is copied to the arrays subi, subj and val which must be allocated to hold at least the number of non-zeros in the matrix.

The data returned must be ordered with `subi` as primary key and `subj` as secondary key.

`subi : int[]`

> Row subscripts are returned in this array.

`subj : int[]`

> Column subscripts are returned in this array.

`cof : double[]`

> Coefficient values are returned in this arary.

5.9.8 `DenseMatrix.getDataAsArray()`

Member of `DenseMatrix`.

`DenseMatrix.getDataAsArray()`

5.9.8.1 **DenseMatrix.getDataAsArray()**

Return the matrix elements as an array. If it is a sparse matrix, it is converted returned as a dense matrix, so the size of the returned array is always exactly the size of the matrix.

returns : `double[]`

5.10 `fusion.Domain`

The `Domain` class defines a set of static method for creating various variable and constraint domains. A `Domain` object specifies a subset of $\mathbb{R}^n$, which can be used to define the feasible domain of variables and expressions.

For further details on the use of these, see

- `Model.variable`

- `Model.constraint`

Static Methods

> `Domain.equalsTo`
>
> `Domain.greaterThan`
>
> `Domain.inPSDCone`
>
> `Domain.inQCone`
>
> `Domain.inRange`
>
> `Domain.inRotatedQCone`

Domain.isInteger
    Creates an object defining the integer domain (all integer values).

Domain.isLinPSD

Domain.isTrilPSD

Domain.lessThan

Domain.unbounded
    Creates an object representing the full euclidian space of any dimension.

5.10.1 Domain.isLinPSD()

Member of Domain.

```
static Domain.isLinPSD()
static Domain.isLinPSD(int n)
static Domain.isLinPSD(int n,int m)
```

5.10.1.1   static Domain.isLinPSD()

returns : PSDDomain

5.10.1.2   static Domain.isLinPSD(n)

n :   int

returns : PSDDomain

5.10.1.3   static Domain.isLinPSD(n,m)

n :   int

m :   int

returns : PSDDomain

5.10.2 Domain.unbounded()

Member of Domain.

```
static Domain.unbounded()
```

5.10.2.1   static Domain.unbounded()

The domain created can be used with constraints and variables of any shape and size.

returns : Domain

5.10.3 `Domain.inPSDCone()`

Member of Domain.

```
static Domain.inPSDCone()
static Domain.inPSDCone(int n)
static Domain.inPSDCone(int n,int m)
```

5.10.3.1 static Domain.inPSDCone()

Creates an object representing the cone of the set:

$$\{X \in R^{n \times n} | 1/2(X + X^T) \in \mathcal{C}_{psd}^n\}$$

The shape of the result is $n \times n$, where n depends on the shape of the variable or constraint with which it is used.

returns : PSDDomain

5.10.3.2 static Domain.inPSDCone(n)

Creates an object representing the cone of the set:

$$\{X \in R^{n \times n} | 1/2(X + X^T) \in \mathcal{C}_{psd}^n\}$$

The shape of the result is $n \times n$.

n : int

returns : PSDDomain

5.10.3.3 static Domain.inPSDCone(n,m)

Creates an object representing m cones of the form:

$$\{X \in R^{n \times n} | 1/2(X + X^T) \in \mathcal{C}_{psd}^n\}$$

The shape of the result is $n \times n \times m$.

n : int

m : int

returns : PSDDomain

5.10.4   `Domain.isTrilPSD()`

Member of `Domain`.

```
static Domain.isTrilPSD()
static Domain.isTrilPSD(int n)
static Domain.isTrilPSD(int n,int m)
```

5.10.4.1   static Domain.isTrilPSD()

Creates an object representing a cone of the form:

$$\{X \in R^{n \times n} | \mathrm{tril}(X) \in \mathcal{C}^n_{psd}\}$$

i.e. the lower triangular part of X define the symmetric matrix that is semidefinite.

The shape of the result is $n \times n \times m$.

 returns : `PSDDomain`

5.10.4.2   static Domain.isTrilPSD(n)

Creates an object representing a cone of the form:

$$\{X \in R^{n \times n} | \mathrm{tril}(X) \in \mathcal{C}^n_{psd}\}$$

i.e. the lower triangular part of X define the symmetric matrix that is semidefinite.

The shape of the result is $n \times n \times m$.

 n :   int

 returns : `PSDDomain`

5.10.4.3   static Domain.isTrilPSD(n,m)

Creates an object representing a cone of the form:

$$\{X \in R^{n \times n} | \mathrm{tril}(X) \in \mathcal{C}^n_{psd}\}$$

i.e. the lower triangular part of X define the symmetric matrix that is semidefinite.

The shape of the result is $n \times n \times m$.

 n :   int

 m :   int

 returns : `PSDDomain`

5.10.5 `Domain.lessThan()`

Member of Domain.

```
static Domain.lessThan(double b)
static Domain.lessThan(double[] bnd)
static Domain.lessThan(Matrix bnd)
```

5.10.5.1 static Domain.lessThan(b)

The object returned can be used with a variable or constraint of any size.

```
b :  double
```
> The upper bound value of the domain.

```
returns : Domain
```

5.10.5.2 static Domain.lessThan(bnd)

The domain created can be used with constraints and variables of one dimension and the same length as the bound.

```
bnd :  double[]
```
> The vector of scalar values defining the bound in each dimension.

```
returns : Domain
```

5.10.5.3 static Domain.lessThan(bnd)

The domain created can be used with constraints and variables of two dimensions and the same shape as the bound.

```
bnd :  Matrix
```
> The upper bound values as a matrix object.

```
returns : Domain
```

5.10.6 `Domain.greaterThan()`

Member of Domain.

```
static Domain.greaterThan(double b)
static Domain.greaterThan(double[] bnd)
static Domain.greaterThan(Matrix bnd)
```

5.10.6.1   static Domain.greaterThan(b)

The object returned can be used with a variable or constraint of any size.

 b :   double
      The lower bound value of the domain.

 returns : Domain

5.10.6.2   static Domain.greaterThan(bnd)

The domain created can be used with constraints and variables of one dimension and the same length as the bound.

 bnd :   double[]
      The vector of scalar values defining the bound in each dimension.

 returns : Domain

5.10.6.3   static Domain.greaterThan(bnd)

The domain created can be used with constraints and variables of two dimensions and the same shape as the bound.

 bnd :   Matrix
      The lower bound values as a matrix object.

 returns : Domain

5.10.7   Domain.equalsTo()

Member of Domain.

```
static Domain.equalsTo(double b)
static Domain.equalsTo(double[] bnd)
static Domain.equalsTo(Matrix bnd)
```

5.10.7.1   static Domain.equalsTo(b)

The object returned can be used with a variable or constraint of any size.

 b :   double
      The value in every dimension.

 returns : Domain

5.10.7.2 static Domain.equalsTo(bnd)

The domain created can be used with constraints and variables of one dimension and the same length as the bound.

bnd : double[]

> The vector of scalar values for each dimension.

returns : Domain

5.10.7.3 static Domain.equalsTo(bnd)

The domain created can be used with constraints and variables of two dimensions and the same shape as the bound.

bnd : Matrix

> The bound values as a matrix object.

returns : Domain

5.10.8 Domain.inQCone()

Member of Domain.

```
static Domain.inQCone()
static Domain.inQCone(int size)
static Domain.inQCone(int m,int n)
```

5.10.8.1 static Domain.inQCone()

Creates a domain representing a quadratic cone of any dimension greater than 1.

returns : Domain

5.10.8.2 static Domain.inQCone(size)

Creates a domain representing a quadratic cone of a given size.

size : int

> The size of the cone, which must be at least 2.

returns : Domain

5.10.8.3   static Domain.inQCone(m,n)

Creates a domain representing the product of a set of identical quadratic cones of a given size.

The result is a domain of shape $m \times n$.

 m :   int
      The number of cones; at least 1.

 n :   int
      The size of each cone; at least 2.

 returns : Domain

5.10.9   Domain.inRange()

Member of Domain.

```
static Domain.inRange(double lb,double ub)
static Domain.inRange(double lb,double[] ub)
static Domain.inRange(double[] lb,double ub)
static Domain.inRange(double[] lb,double[] ub)
static Domain.inRange(double lb,Matrix ub)
static Domain.inRange(Matrix lb,double ub)
static Domain.inRange(Matrix lb,Matrix ub)
```

5.10.9.1   static Domain.inRange(lb,ub)

The created domain can be used with variables or constraints of any size.

 lb :   double
      The lower end of the range.

 ub :   double
      The upper end of the range.

 returns : RangeDomain

5.10.9.2   static Domain.inRange(lb,ub)

The created domain can be used with one-dimensional variables or constraints of a fixed size. The size of the bounds must match the variable or constraint.

 lb :   double
      The lower end of the range in every dimension.

 ub :   double[]
      The upper end of the range in each dimension.

returns : `RangeDomain`

5.10.9.3 static Domain.inRange(lb,ub)

The created domain can be used with one-dimensional variables or constraints of a fixed size. The size of the bounds must match the variable or constraint.

`lb : double[]`
> The lower end of the range in each dimension.

`ub : double`
> The upper end of the range in every dimension.

returns : `RangeDomain`

5.10.9.4 static Domain.inRange(lb,ub)

The created domain can be used with one-dimensional variables or constraints of a fixed size. The size of the bounds must match the variable or constraint.

`lb : double[]`
> The lower end of the range in each dimension.

`ub : double[]`
> The upper end of the range in each dimension.

returns : `RangeDomain`

5.10.9.5 static Domain.inRange(lb,ub)

The created domain can be used with one-dimensional variables or constraints of a fixed size. The size of the bounds must match the variable or constraint.

`lb : double`
> The lower end of the range in every dimension.

`ub : Matrix`
> The upper end of the range in each dimension.

Throws exceptions :

> - `LengthError`

returns : `RangeDomain`

5.10.9.6   static Domain.inRange(lb,ub)

The created domain can be used with one-dimensional variables or constraints of a fixed size. The size of the bounds must match the variable or constraint.

lb :   Matrix

>   The lower end of the range in each dimension.

ub :   double

>   The upper end of the range in every dimension.

Throws exceptions :

- LengthError

returns : RangeDomain

5.10.9.7   static Domain.inRange(lb,ub)

The created domain can be used with one-dimensional variables or constraints of a fixed size. The size of the bounds must match the variable or constraint.

lb :   Matrix

>   The lower end of the range in each dimension.

ub :   Matrix

>   The upper end of the range in each dimension.

returns : RangeDomain

5.10.10   Domain.inRotatedQCone()

Member of Domain.

```
static Domain.inRotatedQCone()
static Domain.inRotatedQCone(int size)
static Domain.inRotatedQCone(int m,int n)
```

5.10.10.1   static Domain.inRotatedQCone()

Creates a domain representing a rotated quadratic cone of any dimension greater than 2.

returns : Domain

5.10.10.2   static Domain.inRotatedQCone(size)

Creates a domain representing a rotated quadratic cone of a given size.

`size :  int`
> The size of the cone; at least 3.

`returns :` `Domain`

5.10.10.3   static Domain.inRotatedQCone(m,n)

Creates a domain representing the product of a set of identical rotated quadratic cones of a given size. The result is a domain of shape $m \times n$.

`m :  int`
> The size of each cone; at least 3.

`n :  int`
> The number of cones; at least 1.

`returns :` `Domain`

5.10.11   Domain.isInteger()

Member of `Domain`.

`static Domain.isInteger()`

5.10.11.1   static Domain.isInteger()

Creates an object defining the integer domain (all integer values).

`returns :` `IntegerDomain`

5.11   fusion.Expr

An `Expr` object represents an expression of the form Ax+b, where A is a matrix on sparse form, x is a variable vector and b is a vector of scalars.

Additionally, the class defines a set of static method for constructing various expressions.

Base class
> `Expression`

Constructors

Expr(long[] ptrb,Variable v,long[] subj,double[] cof,double[] bfix,Set shape,long[] inst)

Construct a new expression from sparse data.

Methods

Expr.eval

Evaluate the expression into simple sparse form.

Expr.numNonzeros

Return the total number of elements in the expression.

Expr.size

Return the total number of elements in the expression.

Static Methods

Expr.add

Construct an expression as the sum items.

Expr.constTerm

Create an expression consisting of a constant vector of values.

Expr.dot

Return an object representing the dot-product of two values.

Expr.flatten

Expr.hstack

Stack a list of expressions horizontally (i.e. in second dimension).

Expr.mul

Multiply two items.

Expr.mulDiag

Compute the diagonal of the product of two matrixes and return it as a vector.

Expr.mulElm

Element-wise multiplication of two items. The two operands must have the same shape.

Expr.neg

Expr.ones

Create a vector of ones as an expression.

Expr.reshape

Reshape the expression into a different shape with the same number of elements.

Expr.stack

Stack expressions in two dimensions.

Expr.sub

Construct an expression as the difference of two items.

Expr.sum

> Sum the elements of an expression

Expr.vstack

> Stack a list of expressions vertically (i.e. in fitst dimension).

Expr.zeros

> Create a vector of zeros as an expression.

Inherited methods

Expression.getModel()

> Return the Model object.

Expression.getShape()

Expression.toString()

> Return a string representation of the expression object.

5.11.1 `Expr.Expr()`

Member of Expr.

`Expr.Expr(long[] ptrb,Variable v,long[] subj,double[] cof,double[] bfix,Set shape,long[] inst)`

5.11.1.1 **Expr.Expr(ptrb,v,subj,cof,bfix,shape,inst)**

Create an expression of the form $A \cdot x + b$, where A is a sparse NxM matrix with P non-zeros, x is a variable and b is a vector of scalars.

The arguments `ptrb_`, `subj_` and `cof_` define the sparse matrix A, and `bfix_` define b.

`ptrb : long[]`

> Array of length N+1 indexes where the A rows begin. The last element is the total number of non-zeros in A, P.

`v : Variable`

> A variable of size M. If $P = 0$, this may be `null`.

`subj : long[]`

> An array of length P. Each element is an index into `v`. If $P = 0$, this may be `null`.

`cof : double[]`

> An array of length P. Each element is a coefficient in the A matrix. If $P = 0$, this may be `null`.

`bfix : double[]`

> An array of length N or `null`. Defines the constant term b.

shape :   Set

> If this is `null`, the expression will be dense and 1-dimensional, and the value of `inst` will be ignored. Otherwise, the parameter defines the number and size of dimensions.

inst :   long[]

> If this is `null`, the expression will be dense, i.e. all elements will be non-zero. Otherwise the array defines the linear indexes of all non-zeros in the set defined by `shape`.

5.11.2 `Expr.reshape()`

Member of Expr.

```
static Expr.reshape(Expression e,Set shp)
static Expr.reshape(Expression e,int size)
static Expr.reshape(Expression e,int dimi,int dimj)
```

Reshape the expression into a different shape with the same number of elements.

e : Expression

> The expression to reshape.

shp : Set

> The new shape of the expression; this must have the same total size as the old shape.

size : int

> Reshape into a one-dimensional expression of this size.

returns : Expression

> A new Expression object.

5.11.2.1   static Expr.reshape(e,shp)

Reshape the expression into a different shape with the same number of elements.

e :   Expression

shp :   Set

returns : Expression

5.11.2.2   static Expr.reshape(e,size)

Reshape the expression into a different shape with the same number of elements.

e :   Expression

size :   int

returns : Expression

5.11.2.3 static Expr.reshape(e,dimi,dimj)

Reshape the expression into a different shape with the same number of elements.

e : Expression

dimi : int

dimj : int

returns : Expression

5.11.3 Expr.sub()

Member of Expr.

```
static Expr.sub(Expression lhs,Matrix rhs)
static Expr.sub(Matrix lhs,Expression rhs)
static Expr.sub(Variable lhs,Matrix rhs)
static Expr.sub(Matrix lhs,Variable rhs)
static Expr.sub(Expression lhs,Expression rhs)
static Expr.sub(Expression lhs,Variable rhs)
static Expr.sub(Variable lhs,Expression rhs)
static Expr.sub(double[] lhs,Variable rhs)
static Expr.sub(Variable lhs,double[] rhs)
static Expr.sub(double[] lhs,Expression rhs)
static Expr.sub(Expression lhs,double[] rhs)
static Expr.sub(Expression lhs,double rhs)
static Expr.sub(double lhs,Expression rhs)
static Expr.sub(Variable lhs,double rhs)
static Expr.sub(double lhs,Variable rhs)
static Expr.sub(double[] lhs,double[] rhs)
static Expr.sub(Variable lhs,Variable rhs)
```

Construct an expression as the difference of two items.

lhs : double
    An expression or variable.

rhs : double
    An expression or variable.

returns : Expression
    A new expression object representing the subtraction of the operands.

5.11.3.1 static Expr.sub(lhs,rhs)

Subtract two expressions.

lhs : Expression
    An expression object.

rhs :   Matrix
     A matrix object.

returns : Expression
     A new expression object representing the difference of the operands.

5.11.3.2 static Expr.sub(lhs,rhs)

Subtract two expressions.

lhs :   Matrix
     A matrix object.

rhs :   Expression
     An expression object.

returns : Expression
     A new expression object representing the difference of the operands.

5.11.3.3 static Expr.sub(lhs,rhs)

Subtract two expressions.

lhs :   Variable
     A matrix object.

rhs :   Matrix
     An variable object.

returns : Expression
     A new expression object representing the difference of the operands.

5.11.3.4 static Expr.sub(lhs,rhs)

Subtract two expressions.

lhs :   Matrix
     A matrix object.

rhs :   Variable
     An variable object.

returns : Expression
     A new expression object representing the difference of the operands.

5.11.3.5   static Expr.sub(lhs,rhs)

Subtract two expressions.

`lhs` :   `Expression`

> An expression object.

`rhs` :   `Expression`

> An expression object.

returns : `Expression`

> A new expression object representing the difference of the operands.

5.11.3.6   static Expr.sub(lhs,rhs)

Subtract two expressions.

`lhs` :   `Expression`

> An expression object.

`rhs` :   `Variable`

> A variable object.

returns : `Expression`

> A new expression object representing the difference of the operands.

5.11.3.7   static Expr.sub(lhs,rhs)

Subtract two expressions.

`lhs` :   `Variable`

> A variable object.

`rhs` :   `Expression`

> An expression object.

returns : `Expression`

> A new expression object representing the difference of the operands.

5.11.3.8   static Expr.sub(lhs,rhs)

Subtract two expressions.

`lhs  :  double[]`

A double array.

`rhs  :  Variable`

A variable object.

`returns : Expression`

A new expression object representing the sum of the operands.

5.11.3.9   static Expr.sub(lhs,rhs)

Subtract two expressions.

`lhs  :  Variable`

A variable object.

`rhs  :  double[]`

A double array.

`returns : Expression`

A new expression object representing the difference of the operands.

5.11.3.10   static Expr.sub(lhs,rhs)

Subtract two expressions.

`lhs  :  double[]`

A double array.

`rhs  :  Expression`

An expression object.

`returns : Expression`

A new expression object representing the difference of the operands.

5.11.3.11   static Expr.sub(lhs,rhs)

Subtract two expressions.

lhs : Expression

>   An expression object.

rhs : double[]

>   A double array.

returns : Expression

>   A new expression object representing the difference of the operands.

5.11.3.12   static Expr.sub(lhs,rhs)

Subtract expression and constant.

lhs : Expression

>   An expression object.

rhs : double

>   A double value.

returns : Expression

>   A new expression object representing the difference of the operands.

5.11.3.13   static Expr.sub(lhs,rhs)

Subtract constant and expression.

lhs : double

>   A double value.

rhs : Expression

>   An expression object.

returns : Expression

>   A new expression object representing the difference of the operands.

5.11.3.14   static Expr.sub(lhs,rhs)

Subtract variable and constant.

 lhs :   Variable
>   A variable object.

 rhs :   double
>   A double value.

 returns : Expression
>   A new expression object representing the difference of the operands.

5.11.3.15   static Expr.sub(lhs,rhs)

Subtract constant and expression.

 lhs :   double
>   A double value.

 rhs :   Variable
>   A variable object.

 returns : Expression
>   A new expression object representing the difference of the operands.

5.11.3.16   static Expr.sub(lhs,rhs)

Subtract two expressions.

 lhs :   double[]
>   A double array.

 rhs :   double[]
>   A double array.

 returns : Expression
>   A new expression object representing the sum of the operands.

5.11.3.17   static Expr.sub(lhs,rhs)

Add two expressions.

 lhs :   Variable

    A variable object.

 rhs :   Variable

    A variable object.

 returns : Expression

    A new expression object representing the difference of the operands.

5.11.4   Expr.hstack()

Member of Expr.

```
static Expr.hstack(Expression[] exprs)
static Expr.hstack(Expression e1,Expression e2)
static Expr.hstack(Expression e1,double e2)
static Expr.hstack(Expression e1,Variable e2)
static Expr.hstack(double e1,Variable e2)
static Expr.hstack(double e1,Expression e2)
static Expr.hstack(Variable e1,double e2)
static Expr.hstack(Variable e1,Variable e2)
static Expr.hstack(Variable e1,Expression e2)
static Expr.hstack(double e1,double e2,Variable e3)
static Expr.hstack(double e1,double e2,Expression e3)
static Expr.hstack(double e1,Variable e2,double e3)
static Expr.hstack(double e1,Variable e2,Variable e3)
static Expr.hstack(double e1,Variable e2,Expression e3)
static Expr.hstack(double e1,Expression e2,double e3)
static Expr.hstack(double e1,Expression e2,Variable e3)
static Expr.hstack(double e1,Expression e2,Expression e3)
static Expr.hstack(Variable e1,double e2,double e3)
static Expr.hstack(Variable e1,double e2,Variable e3)
static Expr.hstack(Variable e1,double e2,Expression e3)
static Expr.hstack(Variable e1,Variable e2,double e3)
static Expr.hstack(Variable e1,Variable e2,Variable e3)
static Expr.hstack(Variable e1,Variable e2,Expression e3)
static Expr.hstack(Variable e1,Expression e2,double e3)
static Expr.hstack(Variable e1,Expression e2,Variable e3)
static Expr.hstack(Variable e1,Expression e2,Expression e3)
static Expr.hstack(Expression e1,double e2,double e3)
static Expr.hstack(Expression e1,double e2,Variable e3)
static Expr.hstack(Expression e1,double e2,Expression e3)
static Expr.hstack(Expression e1,Variable e2,double e3)
static Expr.hstack(Expression e1,Variable e2,Variable e3)
static Expr.hstack(Expression e1,Variable e2,Expression e3)
static Expr.hstack(Expression e1,Expression e2,double e3)
static Expr.hstack(Expression e1,Expression e2,Variable e3)
static Expr.hstack(Expression e1,Expression e2,Expression e3)
```

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 : double

> An expression, a scalar constant or a variable.

e2 : double

> An expression, a scalar constant or a variable.

e3 : double

> An expression, a scalar constant or a variable.

exprs : Expression[]

> A list of expressions.

5.11.4.1   static Expr.hstack(exprs)

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

`exprs :` `Expression[]`

> A list of expressions.

returns : Expression

5.11.4.2 static Expr.hstack(e1,e2)

The expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,1) = dim(e2,1)
dim(e1,3) = dim(e2,3)
...
```

and the dimension of the result is

```
 dim(e1,1)
(dim(e1,2) + dim(e2,2))
 dim(e1,3),
 ...)
```

e1 : `Expression`

    An expression object.

e2 : `Expression`

    An expression object.

returns : `Expression`

5.11.4.3 static Expr.hstack(e1,e2)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 : `Expression`

e2 : `double`

returns : `Expression`

5.11.4.4   static Expr.hstack(e1,e2)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Expression

e2 :   Variable

returns : Expression

5.11.4.5   static Expr.hstack(e1,e2)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   double

e2 :   Variable

returns : Expression

5.11.4.6   static Expr.hstack(e1,e2)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  double
```

```
e2 :  Expression
```

returns : Expression

5.11.4.7   static Expr.hstack(e1,e2)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  Variable
```

```
e2 :  double
```

returns : Expression

5.11.4.8   static Expr.hstack(e1,e2)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   `Variable`

e2 :   `Variable`

returns : `Expression`

5.11.4.9   static Expr.hstack(e1,e2)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   `Variable`

e2 :   `Expression`

returns : `Expression`

5.11.4.10   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   double

e2 :   double

e3 :   `Variable`

returns : `Expression`

5.11.4.11   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   double

e2 :   double

e3 :   `Expression`

returns : Expression

5.11.4.12   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   double
```

```
e2 :   Variable
```

```
e3 :   double
```

returns : Expression

5.11.4.13   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   double
```

```
e2 :   Variable

e3 :   Variable

returns : Expression
```

5.11.4.14   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   double

e2 :   Variable

e3 :   Expression

returns : Expression
```

5.11.4.15   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   double
```

```
e2 :   Expression
```

```
e3 :   double
```

```
returns : Expression
```

5.11.4.16   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
dim(e1,3),
...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   double
```

```
e2 :   Expression
```

```
e3 :   Variable
```

```
returns : Expression
```

5.11.4.17   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
dim(e1,3),
```

```
...)
```

The arguments may be any combination of expressions, scalar constants and variables.

  e1 :   double

  e2 :   Expression

  e3 :   Expression

 returns : Expression

5.11.4.18   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
  dim(e1,1),
 (dim(e1,2) + dim(e2,2) + ...,
  dim(e1,3),
  ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

  e1 :   Variable

  e2 :   double

  e3 :   double

 returns : Expression

5.11.4.19   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
 e1 :  Variable

 e2 :  double

 e3 :  Variable

 returns : Expression
```

5.11.4.20   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
 e1, e2, e3, ...
```

then

```
 dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
 dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
 ...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
 e1 :  Variable

 e2 :  double

 e3 :  Expression

 returns : Expression
```

5.11.4.21   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
 e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
dim(e1,3),
...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Variable

e3 :   double

returns : Expression

5.11.4.22   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
dim(e1,3),
...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Variable

e3 :   Variable

returns : Expression

5.11.4.23   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Variable

e3 :   Expression

returns : Expression

5.11.4.24   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Expression

e3 :   double

returns : Expression

5.11.4.25   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Expression

e3 :   Variable

returns : Expression

5.11.4.26   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Expression

e3 :   Expression

returns : Expression

5.11.4.27   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   Expression
```

```
e2 :   double
```

```
e3 :   double
```

returns : Expression

5.11.4.28   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   Expression
```

```
e2 :  double
```

```
e3 :  Variable
```

returns : Expression

5.11.4.29   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  Expression
```

```
e2 :  double
```

```
e3 :  Expression
```

returns : Expression

5.11.4.30   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  Expression

e2 :  Variable

e3 :  double
```

returns : Expression

5.11.4.31   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  Expression

e2 :  Variable

e3 :  Variable
```

returns : Expression

5.11.4.32   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
```

```
...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 : Expression

e2 : Variable

e3 : Expression

returns : Expression

5.11.4.33   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 : Expression

e2 : Expression

e3 : double

returns : Expression

5.11.4.34   static Expr.hstack(e1,e2,e3)

Stack a list of expressions horizontally (i.e. in second dimension).

All expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2, e3, ...
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1) = ...
dim(e1,3) = dim(e2,3) = dim(e3,3) = ...
...
```

and the dimension of the result is

```
 dim(e1,1),
(dim(e1,2) + dim(e2,2) + ...,
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   `Expression`

e2 :   `Expression`

e3 :   `Variable`

returns : `Expression`

5.11.4.35   static Expr.hstack(e1,e2,e3)

The expressions must have the same shape, except for the second dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,1) = dim(e2,1) = dim(e3,1)
dim(e1,3) = dim(e2,3) = dim(e3,3)
...
```

and the dimension of the result is

```
 dim(e1,1)
(dim(e1,2) + dim(e2,2) + eim(e3,2))
 dim(e1,3),
 ...)
```

e1 :   `Expression`

     An expression object.

e2 :   `Expression`

     An expression object.

e3 :   `Expression`

     An expression object.

returns : `Expression`

5.11.5   `Expr.mulDiag()`

Member of `Expr`.

```
static Expr.mulDiag(Matrix mx,Expression expr)
static Expr.mulDiag(Expression expr,Matrix mx)
static Expr.mulDiag(Matrix expr,Variable v)
```

`static` `Expr.mulDiag(`Variable v`,`Matrix rhs`)`

Compute the diagonal of the product of two matrixes and return it as a vector.

Compute the diagonal of the product of two matrixes, $A \in \mathbb{M}(m,n)$ and $B \in \mathbb{M}(n,p)$. This amounts to a vector $v = (v_1, \ldots, v_n)$ where $v_i = a_{i.}^t b_{.i}$.

expr : Expression

An expression object.

mx : Matrix

An matrix object.

v : Variable

A variable object.

returns : Expression

A new `Expr` object.

5.11.5.1   static Expr.mulDiag(mx,expr)

Compute the diagonal of the product of two matrixes, $A \in \mathbb{M}(m,n)$ and $B \in \mathbb{M}(n,p)$. This amounts to a vector $v = (v_1, \ldots, v_n)$ where $v_i = a_{i.}^t b_{.i}$.

`mx` : `Matrix`

An $m{\times}n$ matrix object.

`expr` : `Expression`

An $n{\times}p$ expression object.

Throws exceptions :

- `LengthError`

returns : Expression

A new `Expr` object.

5.11.5.2   static Expr.mulDiag(expr,mx)

Compute the diagonal of the product of two matrixes, $A \in \mathbb{M}(m,n)$ and $B \in \mathbb{M}(n,p)$. This amounts to a vector $v = (v_1, \ldots, v_n)$ where $v_i = a_{i.}^t b_{.i}$.

`expr` : `Expression`

An $n{\times}p$ expression object.

`mx :   Matrix`

An $m{\times}n$ matrix object.

Throws exceptions :

- `LengthError`

returns : `Expression`

A new `Expr` object.

5.11.5.3   static Expr.mulDiag(expr,v)

Compute the diagonal of the product of two matrixes, $A \in \mathbb{M}(m,n)$ and $B \in \mathbb{M}(n,p)$. This amounts to a vector $v = (v_1, \ldots, v_n)$ where $v_i = a_{i\cdot}^t b_{\cdot i}$.

`expr :   Matrix`

An $m{\times}n$ matrix object.

`v :   Variable`

An $n{\times}p$ variable object.

returns : `Expression`

A new `Expr` object.

5.11.5.4   static Expr.mulDiag(v,rhs)

Compute the diagonal of the product of two matrixes, $A \in \mathbb{M}(m,n)$ and $B \in \mathbb{M}(n,p)$. This amounts to a vector $v = (v_1, \ldots, v_n)$ where $v_i = a_{i\cdot}^t b_{\cdot i}$.

`v :   Variable`

An $n{\times}p$ variable object.

`rhs :   Matrix`

An $m{\times}n$ matrix object.

returns : `Expression`

A new `Expr` object.

5.11.6   Expr.sum()

Member of `Expr`.

```
static Expr.sum(Expression expr)
static Expr.sum(Variable v)
```

Sum the elements of an expression

expr : Expression

> An expression object.

v : Variable

> An variable.

returns : Expression

> An expression object with exactly one element representing the sum of all elements in the operand.

5.11.6.1 static Expr.sum(expr)

Sum the elements of an expression

`expr` : `Expression`

> An expression object.

`returns` : `Expression`

> An expression object with exactly one element representing the sum of all elements in the operand.

5.11.6.2 static Expr.sum(v)

Sum the elements of an expression

`v` : `Variable`

> An variable.

`returns` : `Expression`

> An expression object with exactly one element representing the sum of all elements in the operand.

5.11.7 `Expr.mulElm()`

Member of `Expr`.

```
static Expr.mulElm(double[][] mx,Variable v)
static Expr.mulElm(Variable v,double[][] mx)
static Expr.mulElm(Variable v,double[] a)
static Expr.mulElm(double[] a,Variable v)
static Expr.mulElm(Expression e,double[] a)
static Expr.mulElm(double[] a,Expression e)
static Expr.mulElm(Matrix mx,Variable v)
static Expr.mulElm(Matrix mx,Expression expr)
static Expr.mulElm(Expression expr,Matrix mx)
static Expr.mulElm(Variable v,Matrix mx)
```

Element-wise multiplication of two items. The two operands must have the same shape.

a : double[]

> An coefficeint array.

expr : Expression

> An expression object.

mx : double[][]

> A matrix object.

v : Variable

> A variable object.

returns : Expression

> A two-dimensional expression with the same shape as the operands.

5.11.7.1   static Expr.mulElm(mx,v)

Element-wise multiplication of two items. The two operands must have the same shape.

```
mx :  double[][]
```

```
v :  Variable
```

Throws exceptions :

- LengthError

returns : Expression

5.11.7.2   static Expr.mulElm(v,mx)

Element-wise multiplication of two items. The two operands must have the same shape.

```
v :  Variable
```

```
mx :  double[][]
```

Throws exceptions :

- LengthError

returns : Expression

5.11.7.3 static Expr.mulElm(v,a)

Element-wise multiplication of two items. The two operands must have the same shape.

```
v :    Variable
a :    double[]
returns : Expression
```

5.11.7.4 static Expr.mulElm(a,v)

Element-wise multiplication of two items. The two operands must have the same shape.

```
a :    double[]
v :    Variable
returns : Expression
```

5.11.7.5 static Expr.mulElm(e,a)

Element-wise multiplication of two items. The two operands must have the same shape.

```
e :    Expression
a :    double[]
returns : Expression
```

5.11.7.6 static Expr.mulElm(a,e)

Element-wise multiplication of two items. The two operands must have the same shape.

```
a :    double[]
e :    Expression
returns : Expression
```

5.11.7.7 static Expr.mulElm(mx,v)

Element-wise multiplication of two items. The two operands must have the same shape.

```
mx :    Matrix
v :    Variable
```

Throws exceptions :

- `LengthError`

returns : `Expression`

5.11.7.8   static Expr.mulElm(mx,expr)

Element-wise multiplication of two items. The two operands must have the same shape.

`mx` :   `Matrix`

`expr` :   `Expression`

Throws exceptions :

- `LengthError`

returns : `Expression`

5.11.7.9   static Expr.mulElm(expr,mx)

Element-wise multiplication of two items. The two operands must have the same shape.

`expr` :   `Expression`

`mx` :   `Matrix`

Throws exceptions :

- `LengthError`

returns : `Expression`

5.11.7.10   static Expr.mulElm(v,mx)

Element-wise multiplication of two items. The two operands must have the same shape.

`v` :   `Variable`

`mx` :   `Matrix`

Throws exceptions :

- `LengthError`

returns : `Expression`

5.11.8 `Expr.constTerm()`

Member of Expr.

```
static Expr.constTerm(double[] vals)
static Expr.constTerm(int size,double val)
static Expr.constTerm(double val)
```

Create an expression consisting of a constant vector of values.

size : int

>   Length of the vector

val : double

>   Values to put in vector

returns : Expression

>   An expression representing a vector.

5.11.8.1 static Expr.constTerm(vals)

Create a constant vector of values from val as an expression.

vals :   double[]

returns : Expression

>   An expression representing a vector.

5.11.8.2 static Expr.constTerm(size,val)

Create a constant vector of values from val as an expression.

size :   int

>   Length of the vector

val :   double

>   Value to put in vector

returns : Expression

>   An expression representing a vector.

5.11.8.3 static Expr.constTerm(val)

Create an expression consisting of a constant vector of values.

val :   double

returns : Expression

5.11.9 `Expr.numNonzeros()`

Member of `Expr`.

```
Expr.numNonzeros()
```

5.11.9.1   Expr.numNonzeros()

Return the total number of elements in the expression. Since all expressions has the form $A \cdot x + b$, this corresponds to the height of A.

```
returns : long
```

5.11.10   `Expr.vstack()`

Member of `Expr`.

```
static Expr.vstack(Expression[] exprs)
static Expr.vstack(Expression e1,Expression e2)
static Expr.vstack(Expression e1,Variable e2)
static Expr.vstack(Expression e1,double e2)
static Expr.vstack(Variable e1,Expression e2)
static Expr.vstack(Variable e1,Variable e2)
static Expr.vstack(Variable e1,double e2)
static Expr.vstack(double e1,Expression e2)
static Expr.vstack(double e1,Variable e2)
static Expr.vstack(Expression e1,Expression e2,Expression e3)
static Expr.vstack(Expression e1,Expression e2,Variable e3)
static Expr.vstack(Expression e1,Expression e2,double e3)
static Expr.vstack(Expression e1,Variable e2,Expression e3)
static Expr.vstack(Expression e1,Variable e2,Variable e3)
static Expr.vstack(Expression e1,Variable e2,double e3)
static Expr.vstack(Expression e1,double e2,Expression e3)
static Expr.vstack(Expression e1,double e2,Variable e3)
static Expr.vstack(Expression e1,double e2,double e3)
static Expr.vstack(Variable e1,Expression e2,Expression e3)
static Expr.vstack(Variable e1,Expression e2,Variable e3)
static Expr.vstack(Variable e1,Expression e2,double e3)
static Expr.vstack(Variable e1,Variable e2,Expression e3)
static Expr.vstack(Variable e1,Variable e2,Variable e3)
static Expr.vstack(Variable e1,Variable e2,double e3)
static Expr.vstack(Variable e1,double e2,Expression e3)
static Expr.vstack(Variable e1,double e2,Variable e3)
static Expr.vstack(Variable e1,double e2,double e3)
static Expr.vstack(double e1,Expression e2,Expression e3)
static Expr.vstack(double e1,Expression e2,Variable e3)
static Expr.vstack(double e1,Expression e2,double e3)
static Expr.vstack(double e1,Variable e2,Expression e3)
static Expr.vstack(double e1,Variable e2,Variable e3)
static Expr.vstack(double e1,Variable e2,double e3)
static Expr.vstack(double e1,double e2,Expression e3)
static Expr.vstack(double e1,double e2,Variable e3)
static Expr.vstack(double e1,double e2,double e3)
```

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 : double
> An expression object.

e2 : double
> An expression object.

e3 : double
> An expression object.

exprs : Expression[]
> A list of expressions.

5.11.10.1 static Expr.vstack(exprs)

Stack a list of expressions vertically (i.e. in first dimension).

exprs : Expression[]
> A list of expressions.

returns : Expression

5.11.10.2 static Expr.vstack(e1,e2)

Stack a two expressions vertically (i.e. in first dimension).

e1 : Expression
> An expression object.

e2 : Expression
> An expression object.

returns : Expression

5.11.10.3   static Expr.vstack(e1,e2)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   Expression
```

```
e2 :   Variable
```

```
returns : Expression
```

5.11.10.4   static Expr.vstack(e1,e2)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   Expression
```

```
e2 :   double
```

```
returns : Expression
```

5.11.10.5   static Expr.vstack(e1,e2)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  Variable
```

```
e2 :  Expression
```

```
returns : Expression
```

5.11.10.6   static Expr.vstack(e1,e2)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  Variable
```

```
e2 :  Variable
```

```
returns : Expression
```

5.11.10.7   static Expr.vstack(e1,e2)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   Variable
```

```
e2 :   double
```

```
returns : Expression
```

5.11.10.8   static Expr.vstack(e1,e2)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   double
```

```
e2 :   Expression
```

```
returns : Expression
```

5.11.10.9   static Expr.vstack(e1,e2)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   double

e2 :   Variable

returns : Expression

5.11.10.10   static Expr.vstack(e1,e2,e3)

Stack a three expressions vertically (i.e. in first dimension).

e1 :   Expression

    An expression object.

e2 :   Expression

    An expression object.

e3 :   Expression

    An expression object.

returns : Expression

5.11.10.11   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
```

```
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

  e1 :   Expression

  e2 :   Expression

  e3 :   Variable

  returns : Expression

5.11.10.12   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

  e1 :   Expression

  e2 :   Expression

  e3 :   double

  returns : Expression

5.11.10.13   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

  e1 : Expression

  e2 : Variable

  e3 : Expression

  returns : Expression

5.11.10.14   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

  e1 : Expression

  e2 : Variable

  e3 : Variable

  returns : Expression

5.11.10.15   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

  e1 :   Expression

  e2 :   Variable

  e3 :   double

  returns : Expression

5.11.10.16   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

  e1 :   Expression

  e2 :   double

  e3 :   Expression

  returns : Expression

5.11.10.17   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 : Expression

e2 : double

e3 : Variable

returns : Expression

5.11.10.18   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 : Expression

e2 : double

e3 : double

returns : Expression

5.11.10.19   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Expression

e3 :   Expression

returns : Expression

5.11.10.20   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Expression

e3 :   Variable

returns : Expression

5.11.10.21   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

 e1 :   Variable

 e2 :   Expression

 e3 :   double

 returns : Expression

5.11.10.22   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

 e1 :   Variable

 e2 :   Variable

 e3 :   Expression

 returns : Expression

5.11.10.23   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Variable

e3 :   Variable

returns : Expression

5.11.10.24   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   Variable

e3 :   double

returns : Expression

5.11.10.25   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   double

e3 :   Expression

returns : Expression

5.11.10.26   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   double

e3 :   Variable

returns : Expression

5.11.10.27   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   Variable

e2 :   double

e3 :   double

returns : Expression

5.11.10.28   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

e1 :   double

e2 :   Expression

e3 :   Expression

returns : Expression

5.11.10.29   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   double
```

```
e2 :   Expression
```

```
e3 :   Variable
```

returns : Expression

5.11.10.30   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :   double
```

```
e2 :   Expression
```

```
e3 :   double
```

returns : Expression

5.11.10.31   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  double
```

```
e2 :  Variable
```

```
e3 :  Expression
```

returns : Expression

5.11.10.32   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  double
```

```
e2 :  Variable
```

```
e3 :  Variable
```

returns : Expression

5.11.10.33   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  double
```

```
e2 :  Variable
```

```
e3 :  double
```

returns : Expression

5.11.10.34   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  double
```

```
e2 :  double
```

```
e3 :  Expression
```

returns : Expression

5.11.10.35   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  double
```

```
e2 :  double
```

```
e3 :  Variable
```

```
returns : Expression
```

5.11.10.36   static Expr.vstack(e1,e2,e3)

Stack a list of expressions vertically (i.e. in fitst dimension).

The expressions must have the same shape, except for the first dimension. If expressions are

```
e1, e2
```

then

```
dim(e1,2) = dim(e2,2)
dim(e1,3) = dim(e2,3)
```

and the dimension of the result is

```
(dim(e1,1) + dim(e2,1)
 dim(e1,2),
 dim(e1,3),
 ...)
```

The arguments may be any combination of expressions, scalar constants and variables.

```
e1 :  double
```

```
e2 :  double
```

```
e3 :  double
```

```
returns : Expression
```

5.11.11 `Expr.add()`

Member of `Expr`.

```
static Expr.add(Variable[] vs)
static Expr.add(Expression[] exps)
static Expr.add(Expression lhs,Expression rhs)
static Expr.add(Expression lhs,Variable rhs)
static Expr.add(Variable lhs,Expression rhs)
static Expr.add(Variable lhs,Variable rhs)
static Expr.add(double[] lhs,Variable rhs)
static Expr.add(Variable lhs,double[] rhs)
static Expr.add(double[] lhs,Expression rhs)
static Expr.add(Expression lhs,double[] rhs)
static Expr.add(Expression lhs,double rhs)
static Expr.add(Variable lhs,double rhs)
static Expr.add(double[] lhs,double[] rhs)
static Expr.add(Expression lhs,Matrix rhs)
static Expr.add(Matrix lhs,Expression rhs)
```

Construct an expression as the sum items.

exps : Expression[]

> A list of expressions. All expressions in the array must have the same size. The list must contain at least one element.

lhs : Expression

> An expression or variable.

rhs : double

> An expression or variable.

vs : Variable[]

> A list of Variables. All variables in the array must have the same shape and size. The list must contain at least one element.

returns : Expression

> A new expression object representing the sum of the operands.

5.11.11.1   static Expr.add(vs)

Construct an expression as the sum of a list of variables. The variable shapes must be identical.

The result is a vector expression of the same size as the variables.

vs : `Variable[]`

> A list of Variables. All variables in the array must have the same shape and size. The list must contain at least one element.

returns : `Expression`

5.11.11.2   static Expr.add(exps)

Construct an expression as the sum of a list of expressions. The expression shapes must be identical. The result is an expression with the same shape as each operand.

`exps :` `Expression[]`

> A list of expressions. All expressions in the array must have the same size. The list must contain at least one element.

`returns :` `Expression`

5.11.11.3   static Expr.add(lhs,rhs)

Add two expressions.

`lhs :` `Expression`

> An expression object.

`rhs :` `Expression`

> An expression object.

`returns :` `Expression`

> A new expression object representing the sum of the operands.

5.11.11.4   static Expr.add(lhs,rhs)

Add an expression and a variable.

`lhs :` `Expression`

> An expression object.

`rhs :` `Variable`

> A variable object.

`returns :` `Expression`

> A new expression object representing the sum of the operands.

5.11.11.5   static Expr.add(lhs,rhs)

Add an expression and a variable.

`lhs :` `Variable`

> A variable object.

rhs : `Expression`

> An expression object.

returns : `Expression`

> A new expression object representing the sum of the operands.

5.11.11.6   static Expr.add(lhs,rhs)

Add two variables.

lhs : `Variable`

> A variable object.

rhs : `Variable`

> A variable object.

returns : `Expression`

> A new expression object representing the sum of the operands.

5.11.11.7   static Expr.add(lhs,rhs)

Add a variable and a constant vector.

lhs : `double[]`

> A double array.

rhs : `Variable`

> A variable object. Note that the variable object must be one-dimensional.

returns : `Expression`

> A new expression object representing the sum of the operands.

5.11.11.8   static Expr.add(lhs,rhs)

Add a variable and a constant vector.

lhs : `Variable`

> A variable object. Note that the variable object must be one-dimensional.

rhs : `double[]`

> A double array.

returns : `Expression`

> A new expression object representing the sum of the operands.

5.11.11.9   static Expr.add(lhs,rhs)

Add a constant to an expression.

`lhs` : `double[]`
> A double array.

`rhs` : `Expression`
> An expression object. Note that this must be one-dimensional.

`returns` : `Expression`
> A new expression object representing the sum of the operands.

5.11.11.10   static Expr.add(lhs,rhs)

Add a constant to an expression.

`lhs` : `Expression`
> An expression object. Note that this must be one-dimensional.

`rhs` : `double[]`
> A double array.

`returns` : `Expression`
> A new expression object representing the sum of the operands.

5.11.11.11   static Expr.add(lhs,rhs)

Add expression and constant.

`lhs` : `Expression`
> An expression object.

`rhs` : `double`
> A double value.

`returns` : `Expression`
> A new expression object representing the difference of the operands.

5.11.11.12   static Expr.add(lhs,rhs)

Add variable and constant.

`lhs :` `Variable`

 A variable object.

`rhs :` `double`

 A double value.

returns : `Expression`

 A new expression object representing the difference of the operands.

5.11.11.13   static Expr.add(lhs,rhs)

Add two constant vectors to produce an expression.

`lhs :` `double[]`

 A double array.

`rhs :` `double[]`

 A double array.

returns : `Expression`

 A new expression object representing the sum of the operands.

5.11.11.14   static Expr.add(lhs,rhs)

Add and expression and a matrix.

`lhs :` `Expression`

 An expression object.

`rhs :` `Matrix`

 A matrix object.

returns : `Expression`

 A new expression object representing the sum of the operands.

5.11.11.15   static Expr.add(lhs,rhs)

Add and expression and a matrix.

 lhs :   `Matrix`
     A matrix object.

 rhs :   `Expression`
     An expression object.

 returns : `Expression`
     A new expression object representing the sum of the operands.

5.11.12   Expr.ones()

Member of `Expr`.
 `static Expr.ones(int num)`

5.11.12.1   static Expr.ones(num)

Create a vector of ones as an expression.

 num :   `int`
     The size of the expression.

 returns : `Expression`
     An expression representing a vector of ones.

5.11.13   Expr.zeros()

Member of `Expr`.
 `static Expr.zeros(int num)`

5.11.13.1   static Expr.zeros(num)

Create a vector of zeros as an expression.

 num :   `int`
     The size of the expression.

 returns : `Expression`
     An expression representing a vector of zeros.

5.11.14   Expr.flatten()

Member of Expr.

```
static Expr.flatten(Expression e)
```

5.11.14.1   static Expr.flatten(e)

 e :   Expression

 returns : Expression

5.11.15   Expr.eval()

Member of Expr.

```
Expr.eval()
```

5.11.15.1   Expr.eval()

Evaluate the expression into simple sparse form.

 returns : FlatExpr

The evaluated expression.

5.11.16   Expr.neg()

Member of Expr.

```
static Expr.neg(Expression e)
static Expr.neg(Variable v)
```

5.11.16.1   static Expr.neg(e)

 e :   Expression

 returns : Expression

5.11.16.2   static Expr.neg(v)

 v :   Variable

 returns : Expression

5.11.17   Expr.mul()

Member of Expr.

```
static Expr.mul(Matrix mx,Variable v)
static Expr.mul(Variable v,Matrix mx)
static Expr.mul(Variable v,double[] vals)
static Expr.mul(double[] vals,Variable v)
static Expr.mul(double val,Variable v)
static Expr.mul(Variable v,double val)
static Expr.mul(double[][] mx,Variable v)
static Expr.mul(Variable v,double[][] mx)
static Expr.mul(Expression expr,double val)
static Expr.mul(double val,Expression expr)
static Expr.mul(double[] vals,Expression expr)
static Expr.mul(Expression expr,double[] vals)
static Expr.mul(Expression e,Matrix mx)
static Expr.mul(Matrix mx,Expression e)
```

Multiply two items.

Following combinations of operands are allowed:

A	B
double	Variable
[double]	Expression
[[double]]	
Matrix	

i.e. both `mul(A,B)` and `mul(B,A)` are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

expr : Expression

   An expression object.

mx : double[][]

   A matrix.

v : Variable

   A variable object that may be a scalar or a matrix.

val : double

   A scalar value.

vals : double[]

   A vector of scalars.

5.11.17.1   static Expr.mul(mx,v)

Multiply a matrix with a variable. The variable object may be one- or two-dimensional, and the result is either the one-dimensional matrix/vector product or the two-dimensional matrix/matrix product.

`mx :` `Matrix`

> A matrix object.

`v :` `Variable`

> A variable object.

`returns :` `Expression`

> A new expression object representing the product of the two operands.

5.11.17.2   static Expr.mul(v,mx)

Multiply a matrix with a variable. The variable object may be one- or two-dimensional, and the result is either the one-dimensional matrix/vector product or the two-dimensional matrix/matrix product.

`v :` `Variable`

> A variable object.

`mx :` `Matrix`

> A matrix object.

Throws exceptions :

> • `LengthError`

`returns :` `Expression`

> A new expression object representing the product of the two operands.

5.11.17.3   static Expr.mul(v,vals)

If the `lhs` argument is scalar, the result is a vector. If the `lhs` is a matrix and the dimensions of `lhs` and `rhs` match, then the result is the matrix multiplication.

`v :` `Variable`

> A variable object that may be a scalar or a matrix.

`vals :` `double[]`

> A vector of scalars.

`returns :` `Expression`

5.11.17.4   static Expr.mul(vals,v)

Multiply a variable and a vector of scalars.

`vals :  double[]`
>   A vector of scalars.

`v :  ``Variable`
>   A variable object taht may be a scalar, a vector or a matrix.

`returns : ``Expression`

5.11.17.5   static Expr.mul(val,v)

Multiply a variable by a scalar.

`val :  double`
>   A scalar value.

`v :  ``Variable`
>   A variable object of any shape.

`returns : ``Expression`

5.11.17.6   static Expr.mul(v,val)

Multiply a variable by a scalar.

`v :  ``Variable`
>   A variable object of any shape.

`val :  double`
>   A scalar value.

`returns : ``Expression`

5.11.17.7   static Expr.mul(mx,v)

Multiply two items.

Following combinations of operands are allowed:

A	B
double	Variable
[double]	Expression
[[double]]	
Matrix	

i.e. both `mul(A,B)` and `mul(B,A)` are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

 `mx :  double[][]`

 `v :  ``Variable`

 returns : Expression

5.11.17.8   static Expr.mul(v,mx)

Multiply two items.

Following combinations of operands are allowed:

A	B
double	Variable
[double]	Expression
[[double]]	
Matrix	

i.e. both `mul(A,B)` and `mul(B,A)` are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

 `v :  ``Variable`

 `mx :  double[][]`

 Throws exceptions :

> • LengthError

 returns : Expression

5.11.17.9   static Expr.mul(expr,val)

Multiply an expression by a scalar.

 `expr :  ``Expression`
>   An expression of any shape.

 `val :  double`
>   A scalar value.

 returns : Expression

5.11.17.10   static Expr.mul(val,expr)

Multiply an expression by a scalar.

`val :   double`
>   A scalar value.

`expr :   Expression`
>   An expression of any shape.

`returns : Expression`

5.11.17.11   static Expr.mul(vals,expr)

Multiply a double vector with an expression.

`vals :   double[]`
>   A double vector.

`expr :   Expression`
>   An expression. The shape of the this must match the right-hand side: It must be one- or two-dimensional, and it's dimensions must be matching the left-hand side for multiplication. If this is one-dimensional, the operation is effectively the dot-product.

>   Throws exceptions :

>   - `LengthError`

`returns : Expression`
>   A new expression.

5.11.17.12   static Expr.mul(expr,vals)

Multiply two items.

Following combinations of operands are allowed:

A	B
double	Variable
[double]	Expression
[[double]]	
Matrix	

i.e. both `mul(A,B)` and `mul(B,A)` are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

`expr :` `Expression`

`vals :  double[]`

Throws exceptions :

- `LengthError`

returns : `Expression`

5.11.17.13   static Expr.mul(e,mx)

Multiply two items.

Following combinations of operands are allowed:

A	B
double	Variable
[double]	Expression
[[double]]	
Matrix	

i.e. both `mul(A,B)` and `mul(B,A)` are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

`e :` `Expression`

`mx :` `Matrix`

Throws exceptions :

- `LengthError`

returns : `Expression`

5.11.17.14   static Expr.mul(mx,e)

Multiply two items.

Following combinations of operands are allowed:

A	B
double	Variable
[double]	Expression
[[double]]	
Matrix	

i.e. both `mul(A,B)` and `mul(B,A)` are available.

Note that the size and shape of the operand matter and must adhere to the rules of matrix multiplication.

mx :   Matrix

e :   Expression

Throws exceptions :

- LengthError

returns : Expression

5.11.18   Expr.stack()

Member of Expr.

```
static Expr.stack(Expression[][] exprs)
```

5.11.18.1   static Expr.stack(exprs)

This builds a block matrix if expressions. The height of element in each row must be the same, and the widths of the rows must be the same.

exprs :   Expression[][]

> A list of expressions.

returns : Expression

5.11.19   Expr.dot()

Member of Expr.

```
static Expr.dot(double[] a,Expression expr)
static Expr.dot(Expression expr,double[] a)
static Expr.dot(double[] a,Variable v)
static Expr.dot(Variable v,double[] a)
static Expr.dot(Variable v,Matrix m)
static Expr.dot(Matrix m,Variable v)
```

Return an object representing the dot-product of two values.

Return an object representing the inner product ("dot product") of two vectors, i.e. the sum of the element-wise multiplication.

a : double[]

> A vector of constants

expr : Expression

> A vector expression

v : Variable

> A vector or matrix variable

returns : Expression

> A new expression representing the dot-product between the two operands.

5.11.19.1   static Expr.dot(a,expr)

Return an object representing the dot-product of two values.

Return an object representing the inner product ("dot product") of two vectors, i.e. the sum of the element-wise multiplication.

 `a :   double[]`

 `expr :`   Expression

 returns : Expression

5.11.19.2   static Expr.dot(expr,a)

Return an object representing the dot-product of two values.

Return an object representing the inner product ("dot product") of two vectors, i.e. the sum of the element-wise multiplication.

 `expr :`   Expression

 `a :   double[]`

 returns : Expression

5.11.19.3   static Expr.dot(a,v)

Return an object representing the dot-product of two values.

Return an object representing the inner product ("dot product") of two vectors, i.e. the sum of the element-wise multiplication.

 `a :   double[]`

 `v :`   Variable

 returns : Expression

5.11.19.4   static Expr.dot(v,a)

Return an object representing the dot-product of two values.

Return an object representing the inner product ("dot product") of two vectors, i.e. the sum of the element-wise multiplication.

```
v :   Variable
```

```
a :   double[]
```

returns : Expression

5.11.19.5   static Expr.dot(v,m)

Return an object representing the dot-product of two values.

Return an object representing the inner product ("dot product") of two vectors, i.e. the sum of the element-wise multiplication.

```
v :   Variable
```

```
m :   Matrix
```

Throws exceptions :

- LengthError

returns : Expression

5.11.19.6   static Expr.dot(m,v)

Return an object representing the dot-product of two values.

Return an object representing the inner product ("dot product") of two vectors, i.e. the sum of the element-wise multiplication.

```
m :   Matrix
```

```
v :   Variable
```

Throws exceptions :

- LengthError

returns : Expression

5.11.20   Expr.size()

Member of Expr.

```
Expr.size()
```

5.11.20.1 Expr.size()

Return the total number of elements in the expression. Since all expressions has the form $A \cdot x + b$, this corresponds to the height of A.

returns : `long`

5.12 fusion.Expression

Abstract base class for all objects which can be used as linear expressions of the form Ax+b.

Known direct descendants

Expr

Constructors

Expression(Model m,Set s)

Initialize the expression as belonging to a given model.

Expression(Variable v,Set s)

Methods

Expression.eval

Evaluate the expression into simple sparse form.

Expression.getModel

Return the Model object.

Expression.getShape

Expression.numNonzeros

Return the total number of elements in the expression.

Expression.size

Return the total number of elements in the expression.

Expression.toString

Return a string representation of the expression object.

5.12.1 Expression.Expression()

Member of Expression.

```
Expression.Expression(Model m,Set s)
Expression.Expression(Variable v,Set s)
```

5.12.1.1   Expression.Expression(m,s)

Initialize the expression as belonging to a given model.

m :   Model

  The Model to which the expression belongs.

s :   Set

5.12.1.2   Expression.Expression(v,s)

v :   Variable

s :   Set

5.12.2   Expression.getModel()

Member of Expression.
Expression.getModel()

5.12.2.1   Expression.getModel()

Return the Model object.

returns : Model

  The Model object.

5.12.3   Expression.getShape()

Member of Expression.
Expression.getShape()

5.12.3.1   Expression.getShape()

returns : Set

5.12.4   Expression.toString()

Member of Expression.
Expression.toString()

5.12.4.1 Expression.toString()

Return a string representation of the expression object.

returns : `String`
> A string representation of the object.

5.12.5 Expression.eval()

Member of `Expression`.

`abstract Expression.eval()`

5.12.5.1 abstract Expression.eval()

Evaluate the expression into simple sparse form.

returns : `FlatExpr`
> The evaluated expression.

5.12.6 Expression.numNonzeros()

Member of `Expression`.

`abstract Expression.numNonzeros()`

5.12.6.1 abstract Expression.numNonzeros()

Return the total number of elements in the expression. Since all expressions has the form $A \cdot x + b$, this corresponds to the height of A.

returns : `long`

5.12.7 Expression.size()

Member of `Expression`.

`abstract Expression.size()`

5.12.7.1 abstract Expression.size()

Return the total number of elements in the expression. Since all expressions has the form $A \cdot x + b$, this corresponds to the height of A.

returns : `long`

5.13 `fusion.FlatExpr`

Defines a simple structure containing a sparse representation of a linear expression; basically the result of evaluating an `Expression` object.

Constructors

> `FlatExpr(double[] bfix,long[] ptrb,long[] subj,Variable x,double[] cof,Set shape_,long[] inst_)`
> Construct a new sparse representation of an expression.
>
> `FlatExpr(FlatExpr e)`
> Construct a copy of a `FlatExpr` object.

Methods

> `FlatExpr.size`
> Get the number of non-zero elements in the expression.
>
> `FlatExpr.toString`

5.13.1 `FlatExpr.FlatExpr()`

Member of `FlatExpr`.

`FlatExpr.FlatExpr(double[] bfix,long[] ptrb,long[] subj,Variable x,double[] cof,Set shape_,long[] inst_)`
`FlatExpr.FlatExpr(FlatExpr e)`

5.13.1.1 **FlatExpr.FlatExpr(bfix,ptrb,subj,x,cof,shape_,inst_)**

Construct a new sparse representation of an expression.

`bfix : double[]`
> Constant vector part of the matrix, or `null`.

`ptrb : long[]`
> Array defining each row-offset of the expression.

`subj : long[]`
> Array of column subscripts.

`x : Variable`
> The variable part of the expression.

`cof : double[]`
> Array of matrix coefficients.

`shape_ : Set`

`inst_ : long[]`

5.13.1.2   FlatExpr.FlatExpr(e)

Construct a copy of a FlatExpr object.

    e :   FlatExpr

5.13.2   FlatExpr.toString()

Member of FlatExpr.
 FlatExpr.toString()

5.13.2.1   FlatExpr.toString()

    returns : String

5.13.3   FlatExpr.size()

Member of FlatExpr.
 FlatExpr.size()

5.13.3.1   FlatExpr.size()

Get the number of non-zero elements in the expression.

    returns : int
        The number of non-zero elements in the expression.

5.14   fusion.IntSet

One-dimensional set defined as a range of inregers.

 Base class
        BaseSet

 Constructors

        IntSet(int first,int last)
            Construct set from an integer range.
        IntSet(int length)
            Construct an integer range set of the given length starting at 0.

Methods

> `IntSet.getidx`
>
> `IntSet.getname`
>
> `IntSet.indexToString`
>
> `IntSet.slice`
>
> `IntSet.stride`
>> Return the stride size in the given dimension.

Inherited methods

> `Set.compare(Set other)`
>> Compare two sets and return true if they have the same shape and size.
>
> `BaseSet.dim(int i)`
>> Return the size of the given dimension.
>
> `Set.getSize()`
>> Total number of elements in the set.
>
> `Set.idxtokey(long idx)`
>> Convert a linear index to a N-dimensional key.
>
> `Set.realnd()`
>> Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.
>
> `Set.toString()`
>> Return a string representation of the set.

5.14.1   `IntSet.IntSet()`

Member of `IntSet`.

```
IntSet.IntSet(int first,int last)
IntSet.IntSet(int length)
```

5.14.1.1   IntSet.IntSet(first,last)

Construct set from an integer range.

`first : int`
> First integer in the range.

`last : int`
> Last integer (exclusive) in the range.

5.14.1.2 IntSet.IntSet(length)

Construct an integer range set of the given length starting at 0.

 length : int
>   Length of the range.

5.14.2 `IntSet.stride()`

Member of IntSet.

 `IntSet.stride(int i)`

5.14.2.1 IntSet.stride(i)

Return the stride size in the given dimension.

 i : int
>   Dimension index.

 returns : `long`
>   The stride size in the requested dimension.

5.14.3 `IntSet.getname()`

Member of IntSet.

 `IntSet.getname(long key)`
 `IntSet.getname(int[] key)`

5.14.3.1 IntSet.getname(key)

Return a string representing the index.

 key : long
>   A linear index.

 returns : `String`
>   Get a string representing the item identified by the key.

5.14.3.2 IntSet.getname(key)

Return a string representing the index.

`key :  int[]`

   An N-dimensional index.

`returns : String`

   Get a string representing the item identified by the key.

5.14.4 `IntSet.slice()`

Member of `IntSet`.

`IntSet.slice(int firstidx,int lastidx)`
`IntSet.slice(int[] firstidx,int[] lastidx)`

5.14.4.1 **IntSet.slice(firstidx,lastidx)**

Create a set object representing a slice of this set.

`firstidx :  int`

   Last index in the range.

`lastidx :  int`

`returns : Set`

   A new `Set` object representing the slice.

5.14.4.2 **IntSet.slice(firstidx,lastidx)**

Create a set object representing a slice of this set.

`firstidx :  int[]`

   Last index in each dimension in the range.

`lastidx :  int[]`

`returns : Set`

   A new `Set` object representing the slice.

5.14.5 `IntSet.indexToString()`

Member of `IntSet`.

`IntSet.indexToString(long index)`

5.14.5.1 IntSet.indexToString(index)

```
index  :   long
```

```
returns : String
```

5.14.6 IntSet.getidx()

Member of `IntSet`.
```
IntSet.getidx(int key)
```

5.14.6.1 IntSet.getidx(key)

```
key  :   int
```

```
returns : int
```

5.15 `fusion.IntegerConicVariable`

This class represents a conic integer constrained variable.

The variable never has any dual solution data, and it is always an error to request it.

Base class

`ConicVariable`

Inherited methods

`Variable.antidiag(int index)`
Return an anti-diagonal of a square matrix.

`Variable.antidiag()`
Return the anti-diagonal of a square matrix

`Variable.asExpr()`
Create an expression corresponding to the variable object.

`Variable.compress()`
Reshape a variable object by removing all dimensions of size 1.

`Variable.diag()`
Return the diagonal of a square matrix

`Variable.diag(int index)`
Return a diagonal of a square matrix.

`Variable.dual(int[] firstidx,int[] lastidx)`
Return a slice of the dual solution.

`Variable.dual(int firstidx,int lastidx)`
  Return a slice of the dual solution.

`Variable.dual()`
  Get the dual solution value of the variable.

`Variable.index(int[] idx)`
  Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1)`
  Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1,int i2)`
  Get a single element from a one-dimensional variable.

`Variable.index(int i0)`
  Get a single element from a one-dimensional variable.

`Variable.index_flat(long i)`
  Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`
  Return a single value of the primal solution.

`Variable.level()`
  Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`
  Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`
  Return a slice of the primal solution.

`Variable.pick(int[] idxs)`
  Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`
  Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`
  Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`
  Get the number of elements in the variable.

`ModelVariable.slice(int[] first,int[] last)`
  Get a multi-dimensional slice of the variable.

`ModelVariable.slice(int first,int last)`
  Create a variable object representing a slice of the variable.

`ModelVariable.toString()`
  Create a string-representation of the variable.

`Variable.transpose()`
  Transpose a vector or matrix variable

5.16 fusion.IntegerDomain

The class can only be instantiated through Domain.isInteger.

5.17 fusion.IntegerLinearVariable

This class represents an integer constrained variable which is either free, bounded below or bounded above.

The variable never has any dual solution data, and it is always an error to request it.

Base class

LinearVariable

Inherited methods

Variable.antidiag(int index)
    Return an anti-diagonal of a square matrix.

Variable.antidiag()
    Return the anti-diagonal of a square matrix

Variable.asExpr()
    Create an expression corresponding to the variable object.

Variable.compress()
    Reshape a variable object by removing all dimensions of size 1.

Variable.diag()
    Return the diagonal of a square matrix

Variable.diag(int index)
    Return a diagonal of a square matrix.

Variable.dual(int[] firstidx,int[] lastidx)
    Return a slice of the dual solution.

Variable.dual(int firstidx,int lastidx)
    Return a slice of the dual solution.

Variable.dual()
    Get the dual solution value of the variable.

Variable.index(int[] idx)
    Get a single element from a one-dimensional variable.

Variable.index(int i0,int i1)
    Get a single element from a one-dimensional variable.

Variable.index(int i0,int i1,int i2)
    Get a single element from a one-dimensional variable.

`Variable.index(int i0)`

Get a single element from a one-dimensional variable.

`Variable.index_flat(long i)`

Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`

Return a single value of the primal solution.

`Variable.level()`

Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`

Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`

Return a slice of the primal solution.

`Variable.pick(int[] idxs)`

Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`

Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`

Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`

Get the number of elements in the variable.

`ModelVariable.slice(int[] first,int[] last)`

Get a multi-dimensional slice of the variable.

`ModelVariable.slice(int first,int last)`

Create a variable object representing a slice of the variable.

`ModelVariable.toString()`

Create a string-representation of the variable.

`Variable.transpose()`

Transpose a vector or matrix variable

5.18   fusion.IntegerRangedVariable

This class represents a ranged integer constrained variable.

The variable never has any dual solution data, and it is always an error to request it.

Base class

`RangedVariable`

Inherited methods

> `Variable.antidiag(int index)`
>> Return an anti-diagonal of a square matrix.
>
> `Variable.antidiag()`
>> Return the anti-diagonal of a square matrix
>
> `Variable.asExpr()`
>> Create an expression corresponding to the variable object.
>
> `Variable.compress()`
>> Reshape a variable object by removing all dimensions of size 1.
>
> `Variable.diag()`
>> Return the diagonal of a square matrix
>
> `Variable.diag(int index)`
>> Return a diagonal of a square matrix.
>
> `Variable.dual(int[] firstidx,int[] lastidx)`
>> Return a slice of the dual solution.
>
> `Variable.dual(int firstidx,int lastidx)`
>> Return a slice of the dual solution.
>
> `Variable.dual()`
>> Get the dual solution value of the variable.
>
> `Variable.index(int[] idx)`
>> Get a single element from a one-dimensional variable.
>
> `Variable.index(int i0,int i1)`
>> Get a single element from a one-dimensional variable.
>
> `Variable.index(int i0,int i1,int i2)`
>> Get a single element from a one-dimensional variable.
>
> `Variable.index(int i0)`
>> Get a single element from a one-dimensional variable.
>
> `Variable.index_flat(long i)`
>> Index into the variable as if it was a one-dimensional object.
>
> `Variable.level(int index)`
>> Return a single value of the primal solution.
>
> `Variable.level()`
>> Get the primal solution value of the variable.
>
> `Variable.level(int[] firstidx,int[] lastidx)`
>> Return a slice of the primal solution.
>
> `Variable.level(int firstidx,int lastidx)`
>> Return a slice of the primal solution.

RangedVariable.lowerBoundVar()

> Get a variable object corresponding to the lower bound of the ranged variable.

Variable.pick(int[] idxs)

> Create a vector-variable by picking a list of indexes from this variable.

Variable.pick(int[][] midxs)

> Create a matrix-variable by picking a list of indexes from this variable.

Variable.pick_flat(long[] indexes)

> Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

Variable.size()

> Get the number of elements in the variable.

ModelVariable.slice(int[] first,int[] last)

> Get a multi-dimensional slice of the variable.

ModelVariable.slice(int first,int last)

> Create a variable object representing a slice of the variable.

ModelVariable.toString()

> Create a string-representation of the variable.

Variable.transpose()

> Transpose a vector or matrix variable

RangedVariable.upperBoundVar()

> Get a variable object corresponding to the upper bound of the ranged variable.

5.19 fusion.LinearConstraint

A linear constraint defines a block of constraints with the same linear domain. The domain is either a product of product of one-dimensional half-spaces (linear inequalities), a fixed value vector (equalities) or the whole space (free constraints).

The *type* of a linear variable is immutable; it is either free, an inequality or an equality, but the linear expression and the right-hand side can be modified.

The class is not meant to be instantiated directly, but must be created by calling the Model.variable method.

Base class

> ModelConstraint

Inherited methods

> Constraint.add(Expression expr)
>
> > Add an expression to the constraint expression.

Constraint.add(Variable v)

    Add an expression to the constraint expression.

Constraint.dual()

    Get the dual solution value of the variable.

Constraint.dual(int[] firstidx,int[] lastidx)

    Return a slice of the dual solution.

Constraint.dual(int firstidx,int lastidx)

    Return a slice of the dual solution.

Constraint.get_model()

    Get the original model object.

Constraint.get_nd()

    Get the number of dimensions of the constraint.

Constraint.index(int[] idx)

    Get a single element from a one-dimensional constraint.

Constraint.index(int idx)

    Get a single element from a one-dimensional constraint.

Constraint.level(int index)

    Return a single value of the primal solution.

Constraint.level(int firstidx,int lastidx)

    Return a slice of the primal solution.

Constraint.level()

    Get the primal solution value of the variable.

Constraint.level(int[] firstidx,int[] lastidx)

    Return a slice of the primal solution.

Constraint.size()

    Get the total number of elements in the constraint.

ModelConstraint.slice(int first,int last)

    Get a slice of the constraint.

ModelConstraint.slice(int[] first,int[] last)

    Get a multi-dimensional slice of the constraint.

ModelConstraint.toString()

5.20 fusion.LinearVariable

A linear variable defines a block of variables with the same linear domain. The domain is either a product of product of one-dimensional half-spaces (linear inequalities), a fixed value vector (equalities) or the whole space (free variables).

The *type* of a linear variable is immutable; it is either free, an inequality or an equality.

The class is not meant to be instantiated directly, but must be created by calling the `Model.variable` method.

Base class

>   `ModelVariable`

Known direct descendants

>   `IntegerLinearVariable`

Inherited methods

>   `Variable.antidiag(int index)`
>
>> Return an anti-diagonal of a square matrix.
>
>   `Variable.antidiag()`
>
>> Return the anti-diagonal of a square matrix
>
>   `Variable.asExpr()`
>
>> Create an expression corresponding to the variable object.
>
>   `Variable.compress()`
>
>> Reshape a variable object by removing all dimensions of size 1.
>
>   `Variable.diag()`
>
>> Return the diagonal of a square matrix
>
>   `Variable.diag(int index)`
>
>> Return a diagonal of a square matrix.
>
>   `Variable.dual(int[] firstidx,int[] lastidx)`
>
>> Return a slice of the dual solution.
>
>   `Variable.dual(int firstidx,int lastidx)`
>
>> Return a slice of the dual solution.
>
>   `Variable.dual()`
>
>> Get the dual solution value of the variable.
>
>   `Variable.index(int[] idx)`
>
>> Get a single element from a one-dimensional variable.
>
>   `Variable.index(int i0,int i1)`
>
>> Get a single element from a one-dimensional variable.
>
>   `Variable.index(int i0,int i1,int i2)`
>
>> Get a single element from a one-dimensional variable.
>
>   `Variable.index(int i0)`
>
>> Get a single element from a one-dimensional variable.
>
>   `Variable.index_flat(long i)`
>
>> Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`

Return a single value of the primal solution.

`Variable.level()`

Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`

Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`

Return a slice of the primal solution.

`Variable.pick(int[] idxs)`

Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`

Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`

Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`

Get the number of elements in the variable.

`ModelVariable.slice(int[] first,int[] last)`

Get a multi-dimensional slice of the variable.

`ModelVariable.slice(int first,int last)`

Create a variable object representing a slice of the variable.

`ModelVariable.toString()`

Create a string-representation of the variable.

`Variable.transpose()`

Transpose a vector or matrix variable

5.21 `fusion.Matrix`

Base class for all matrix objects.

Known direct descendants

`DenseMatrix`, `SparseMatrix`

Methods

`Matrix.get`

`Matrix.getDataAsArray`

Return the data a dense array of values.

Matrix.getDataAsTriplets
:   Return the matrix data in triplet format.

Matrix.isSparse
:   Returns true if the matrix is sparse.

Matrix.numColumns
:   Returns the number columns in the matrix.

Matrix.numNonzeros
:   Returns the number of non-zeros in the matrix.

Matrix.numRows
:   Returns the number rows in the matrix.

Matrix.toString
:   Get a string representation of the matrix.

Matrix.transpose
:   Transpose the matrix.

Static Methods

Matrix.antidiag
:   Create a sparse square matrix with a given vector as anti-diagonal

Matrix.diag
:   Create a sparse square matrix with a given vector as diagonal

Matrix.sparse

5.21.1 Matrix.isSparse()

Member of Matrix.

```
 abstract Matrix.isSparse()
```

5.21.1.1   abstract Matrix.isSparse()

Returns true if the matrix is sparse.

```
 returns : boolean
```

5.21.2 Matrix.numColumns()

Member of Matrix.

```
 Matrix.numColumns()
```

5.21.2.1 Matrix.numColumns()

Returns the number columns in the matrix.

returns : `int`

> The number of columns.

5.21.3 `Matrix.get()`

Member of `Matrix`.

```
abstract Matrix.get(int i,int j)
```

5.21.3.1 abstract Matrix.get(i,j)

i : `int`

j : `int`

returns : `double`

5.21.4 `Matrix.diag()`

Member of `Matrix`.

```
static Matrix.diag(double[] d)
static Matrix.diag(double[] d,int k)
static Matrix.diag(int n,double val)
static Matrix.diag(int n,double val,int k)
static Matrix.diag(Matrix[] md)
static Matrix.diag(int num,Matrix mv)
```

Create a sparse square matrix with a given vector as diagonal

d : double[]

> The diagonal vector

k : int

> The diagonal index. k=0 is the default and means the main diagonal. k¿0 means diagonals above the main, and k¡0 means the diagonals below the main.

md : Matrix[]

> A list of square matrixes that are used to create a block-diagonal square matrix.

n : int

> The size of each side in the matrix.

val : double

> Use this value for all diagonal elements.

5.21.4.1   static Matrix.diag(d)

Create a sparse square matrix with a given vector as diagonal

```
d :   double[]
```

```
returns : Matrix
```

5.21.4.2   static Matrix.diag(d,k)

Create a sparse square matrix with a given vector as diagonal

```
d :   double[]
```

```
k :   int
```

```
returns : Matrix
```

5.21.4.3   static Matrix.diag(n,val)

Create a sparse square matrix with a given vector as diagonal

```
n :   int
```

```
val :   double
```

```
returns : Matrix
```

5.21.4.4   static Matrix.diag(n,val,k)

Create a sparse square matrix with a given vector as diagonal

```
n :   int
```

```
val :   double
```

```
k :   int
```

```
returns : Matrix
```

5.21.4.5   static Matrix.diag(md)

Construct a sparse matrix matrix with non-overlapping blocks.

Note that the blocks are not required to be square, so the resulting matrix is not necessarily square.

```
md :   Matrix[]
```
A list of square matrixes that are used to create a block-diagonal square matrix.

Throws exceptions :

- `LengthError`

returns : `Matrix`

A sparse block diagonal matrix.

5.21.4.6   static Matrix.diag(num,mv)

Create a sparse matrix by placing a matrix `num` times after itself. If the `mv` is square the result will be a square block-diagonal sparse matrix.

`num  :  int`

Number of times to repeat the `mv` matrix.

`mv  :  Matrix`

Throws exceptions :

- `LengthError`

returns : `Matrix`

5.21.5   `Matrix.transpose()`

Member of `Matrix`.

`abstract Matrix.transpose()`

5.21.5.1   abstract Matrix.transpose()

Transpose the matrix.

returns : `Matrix`

5.21.6   `Matrix.getDataAsArray()`

Member of `Matrix`.

`abstract Matrix.getDataAsArray()`

5.21.6.1   abstract Matrix.getDataAsArray()

Return the matrix elements as an array. If it is a sparse matrix, it is converted returned as a dense matrix, so the size of the returned array is always exactly the size of the matrix.

returns : `double[]`

5.21.7 `Matrix.numRows()`

Member of `Matrix`.

`Matrix.numRows()`

5.21.7.1 **Matrix.numRows()**

Returns the number rows in the matrix.

returns : `int`

>   The number of rows.

5.21.8 `Matrix.toString()`

Member of `Matrix`.

`abstract Matrix.toString()`

5.21.8.1 **abstract Matrix.toString()**

Get a string representation of the matrix.

returns : `String`

>   A string representation of the matrix.

5.21.9 `Matrix.antidiag()`

Member of `Matrix`.

```
static Matrix.antidiag(double[] d)
static Matrix.antidiag(double[] d,int k)
static Matrix.antidiag(int n,double val)
static Matrix.antidiag(int n,double val,int k)
```

Create a sparse square matrix with a given vector as anti-diagonal

d : double[]

>   The diagonal vector

k : int

>   The diagonal index. k=0 is the default and means the main diagonal. k¿0 means diagonals above
>   the main, and k¡0 means the diagonals below the main.

n : int

>   The size of each side in the matrix.

val : double

> Use this value for all diagonal elements.

5.21.9.1   static Matrix.antidiag(d)

Create a sparse square matrix with a given vector as anti-diagonal

```
d :   double[]
```

returns : Matrix

5.21.9.2   static Matrix.antidiag(d,k)

Create a sparse square matrix with a given vector as anti-diagonal

```
d :   double[]
```

```
k :   int
```

returns : Matrix

5.21.9.3   static Matrix.antidiag(n,val)

Create a sparse square matrix with a given vector as anti-diagonal

```
n :   int
```

```
val :   double
```

returns : Matrix

5.21.9.4   static Matrix.antidiag(n,val,k)

Create a sparse square matrix with a given vector as anti-diagonal

```
n :   int
```

```
val :   double
```

```
k :   int
```

returns : Matrix

5.21.10   Matrix.sparse()

Member of Matrix.

```
static Matrix.sparse(int nrow,int ncol,int[] subi,int[] subj,double[] val)
static Matrix.sparse(int[] subi,int[] subj,double[] val)
static Matrix.sparse(int[] subi,int[] subj,double val)
static Matrix.sparse(int nrow,int ncol,int[] subi,int[] subj,double val)
static Matrix.sparse(int nrow,int ncol)
static Matrix.sparse(double[][] data)
static Matrix.sparse(Matrix[][] blocks)
static Matrix.sparse(Matrix v)
```

5.21.10.1   static Matrix.sparse(nrow,ncol,subi,subj,val)

Construct a sparse matrix from triplet-format data.

nrow :   int

Number of rows.

ncol :   int

Number of columns.

subi :   int[]

Row indexes of the non-zero elements.

subj :   int[]

Column indexes of the non-zero elements.

val :   double[]

Values of the non-zero elements

returns : Matrix

A sparse matrix object.

5.21.10.2   static Matrix.sparse(subi,subj,val)

Construct a sparse matrix from triplet-format data.

subi :   int[]

Row indexes of the non-zero elements.

subj :   int[]

Column indexes of the non-zero elements.

val :   double[]

Values of the non-zero elements

returns : `Matrix`

    A sparse matrix object.

5.21.10.3   static Matrix.sparse(subi,subj,val)

Construct a sparse matrix from triplet-format data.

`subi : int[]`

    Row indexes of the non-zero elements.

`subj : int[]`

    Column indexes of the non-zero elements.

`val : double`

    A single value used for all non-zero elements

returns : `Matrix`

    A sparse matrix object.

5.21.10.4   static Matrix.sparse(nrow,ncol,subi,subj,val)

Construct a sparse matrix from triplet-format data.

`nrow : int`

    Number of rows.

`ncol : int`

    Number of columns.

`subi : int[]`

    Row indexes of the non-zero elements.

`subj : int[]`

    Column indexes of the non-zero elements.

`val : double`

    A single value used for all non-zero elements

returns : `Matrix`

    A sparse matrix object.

5.21.10.5   static Matrix.sparse(nrow,ncol)

Construct an empty sparse matrix.

`nrow :  int`

>   Number of rows in the matrix.

`ncol :  int`

>   Number of columns in the matrix.

returns : `Matrix`

>   A sparse matrix object.

5.21.10.6   static Matrix.sparse(data)

Construct a sparse matrix from dense data

`data :  double[][]`

>   A two-dimensional array of values.

returns : `Matrix`

>   A sparse matrix object.

5.21.10.7   static Matrix.sparse(blocks)

Construct a sparse matrix from blocks.

`blocks :  Matrix[][]`

>   The matrix data. This is a two-dimensional array of `Matrix` objects; each entry must be a
>   `DenseMatrix`, a `SparseMatrix` or `null`. In `blocks`, all elements in a row must have the same
>   height, and all elements in a column must have the same width.

>   Entries that are `null` will be interpreted as a block of zeros whose height and width are deduced
>   from the other elements in the same row and column. Any row that contains only `null` entries
>   will have height 0, and any column that contains only `null` entries will have width 0.

returns : `Matrix`

>   A sparse matrix object.

5.21.10.8   static Matrix.sparse(v)

Construct a sparse matrix from another `Matrix` object.

`v :  Matrix`

>   A matrix to construct a sparse matrix from.

returns : `Matrix`

> A sparse matrix object.

5.21.11   `Matrix.getDataAsTriplets()`

Member of `Matrix`.

 abstract `Matrix.getDataAsTriplets`(int[] subi,int[] subj,double[] val)

5.21.11.1   abstract Matrix.getDataAsTriplets(subi,subj,val)

Return the matrix data in triplet format. Data is copied to the arrays subi, subj and val which must
be allocated to hold at least the number of non-zeros in the matrix.

The data returned must be ordered with `subi` as primary key and `subj` as secondary key.

 `subi :  int[]`

> Row subscripts are returned in this array.

 `subj :  int[]`

> Column subscripts are returned in this array.

 `val :   double[]`

> Coefficient values are returned in this arary.

 Throws exceptions :

> - `LengthError`

5.21.12   `Matrix.numNonzeros()`

Member of `Matrix`.

 abstract `Matrix.numNonzeros`()

5.21.12.1   abstract Matrix.numNonzeros()

Returns the number of non-zeros in the matrix.

 returns : `long`

> The number of non-zeros.

5.22   `fusion.Model`

The object containing all data related to a single optimization model.

Constructors

> `Model(String name,String licfile)`
>> Create a new model object.
>
> `Model(String name)`
>> Create a new named model object.
>
> `Model()`
>> Create a new unnamed model object.

Methods

> `Model.acceptedSolutionStatus`
>> Get or set the accepted solution status.
>
> `Model.constraint`
>> Create a new constraint in the model.
>
> `Model.dualObjValue`
>> Get the dual objective value.
>
> `Model.getConstraint`
>> Get the constraint corresponding to the given name
>
> `Model.getDualSolutionStatus`
>> Return the status of the dual solution.
>
> `Model.getPrimalSolutionStatus`
>> Return the status of the primal solution.
>
> `Model.getSolverDoubleInfo`
>> Fetch a solution information item from the solver
>
> `Model.getSolverIntInfo`
>> Fetch a solution information item from the solver
>
> `Model.getSolverLIntInfo`
>> Fetch a solution information item from the solver
>
> `Model.getVariable`
>> Get the variable corresponding to the given name
>
> `Model.objective`
>> Replace the objective expression.
>
> `Model.primalObjValue`
>> Get the primal objective value.

Model.setLogHandler
>   Attach a log handler.

Model.setSolverParam
>   Set a solver parameter

Model.solve
>   Attempt to optimize the model.

Model.sparseVariable
>   Create a new sparse variable in the model.

Model.variable
>   Create a new variable in the model.

Model.writeTask
>   Dump the current solver task to a file.

5.22.1 Model.Model()

Member of Model.

```
Model.Model(String name,String licfile)
Model.Model(String name)
Model.Model()
```

5.22.1.1 Model.Model(name,licfile)

Create a new model object.

name : String
>   The name of the model. If this is null, the model is given no name.

licfile : String
>   License file name.

5.22.1.2 Model.Model(name)

Create a new named model object.

name : String
>   The name of the model.

5.22.1.3 Model.Model()

Create a new unnamed model object.

5.22.2   `Model.acceptedSolutionStatus()`

Member of `Model`.

```
Model.acceptedSolutionStatus(AccSolutionStatus what)
Model.acceptedSolutionStatus()
```

Get or set the accepted solution status.

This sets or gets the flag that indicated what solutions are accepted as *expected* when fetching primal and dual solution values.

When fetching a solution value the status of the solution is checked against the flag. If it matches, the solution is returned, otherwise an exception is thrown. The two methods `Model.getPrimalSolutionStatus` and `Model.getDualSolutionStatus` can be used to get the *actual* status of the solutions.

By default the accepted solution status is `AccSolutionStatus.NearOptimal`.

5.22.2.1   Model.acceptedSolutionStatus(what)

This defines which solution status values are accepted when fetching solution values. Requesting a solution value for a variable or constraint whose status does not match the accepted status will cause an error.

See `AccSolutionStatus` for details on status matching.

  `what` :   `AccSolutionStatus`
       The new accepted solution status.

5.22.2.2   Model.acceptedSolutionStatus()

Get the the accepted solution status.

  returns : `AccSolutionStatus`

5.22.3   `Model.getPrimalSolutionStatus()`

Member of `Model`.

```
Model.getPrimalSolutionStatus()
```

5.22.3.1   Model.getPrimalSolutionStatus()

Return the status of the primal solution.

  returns : `SolutionStatus`

5.22.4 `Model.getConstraint()`

Member of `Model`.

`Model.getConstraint`(String name)

5.22.4.1 Model.getConstraint(name)

Get the constraint corresponding to the given name

name : **String**

> The constraint name.

returns : `Constraint`

> The constraint object with the given name.

5.22.5 `Model.constraint()`

Member of `Model`.

`Model.constraint`(String name,`Expression` expr,`PSDDomain` dom)
`Model.constraint`(`Expression` expr,`PSDDomain` dom)
`Model.constraint`(String name,`Set` shape,`Expression` expr,`Domain` dom)
`Model.constraint`(`Set` shape,`Expression` expr,`Domain` dom)
`Model.constraint`(String name,`Expression` expr,`Domain` dom)
`Model.constraint`(`Expression` expr,`Domain` dom)
`Model.constraint`(String name,`Set` shape,`Expression` expr,`RangeDomain` dom)
`Model.constraint`(`Set` shape,`Expression` expr,`RangeDomain` dom)
`Model.constraint`(String name,`Expression` expr,`RangeDomain` dom)
`Model.constraint`(`Expression` expr,`RangeDomain` dom)
`Model.constraint`(String name,`Variable` v,`PSDDomain` dom)
`Model.constraint`(`Variable` v,`PSDDomain` dom)
`Model.constraint`(String name,`Set` shape,`Variable` v,`Domain` dom)
`Model.constraint`(`Set` shape,`Variable` v,`Domain` dom)
`Model.constraint`(String name,`Variable` v,`Domain` dom)
`Model.constraint`(`Variable` v,`Domain` dom)
`Model.constraint`(String name,`Set` shape,`Variable` v,`RangeDomain` dom)
`Model.constraint`(`Set` shape,`Variable` v,`RangeDomain` dom)
`Model.constraint`(String name,`Variable` v,`RangeDomain` dom)
`Model.constraint`(`Variable` v,`RangeDomain` dom)

Create a new constraint in the model.

dom : Domain

> Defines the domain of the expression. The shape and size of the domain must match the shape og the expression.

expr : Expression

> An expression.

name : String

> Name of the constraint. This must be unique among all constraints in the model. The value `null` is allowed instead of a unique name.

shape : Set

> Defines the shape of the constraint. If this is `null`, the shape will be derived from the shape of `expr`.

v : Variable

> A variable used as an expression.

returns : Constraint

> The newly created constraint.

5.22.5.1   Model.constraint(name,expr,dom)

Create a new constraint in the model.

```
name :   String

expr :   Expression

dom  :   PSDDomain

returns : Constraint
```

5.22.5.2   Model.constraint(expr,dom)

Create a new constraint in the model.

```
expr :   Expression

dom  :   PSDDomain

returns : Constraint
```

5.22.5.3   Model.constraint(name,shape,expr,dom)

Create a new constraint in the model.

```
name :   String

shape :   Set

expr :   Expression

dom  :   Domain

returns : Constraint
```

5.22.5.4 Model.constraint(shape,expr,dom)

Create a new constraint in the model.

```
shape :   Set

expr :   Expression

dom :   Domain

returns : Constraint
```

5.22.5.5 Model.constraint(name,expr,dom)

Create a new constraint in the model.

```
name :   String

expr :   Expression

dom :   Domain

returns : Constraint
```

5.22.5.6 Model.constraint(expr,dom)

Create a new constraint in the model.

```
expr :   Expression

dom :   Domain

returns : Constraint
```

5.22.5.7 Model.constraint(name,shape,expr,dom)

Create a new constraint in the model.

```
name :   String

shape :   Set

expr :   Expression

dom :   RangeDomain

returns : Constraint
```

5.22.5.8   Model.constraint(shape,expr,dom)

Create a new constraint in the model.

```
shape :   Set

expr :   Expression

dom :   RangeDomain

returns : Constraint
```

5.22.5.9   Model.constraint(name,expr,dom)

Create a new constraint in the model.

```
name :   String

expr :   Expression

dom :   RangeDomain

returns : Constraint
```

5.22.5.10   Model.constraint(expr,dom)

Create a new constraint in the model.

```
expr :   Expression

dom :   RangeDomain

returns : Constraint
```

5.22.5.11   Model.constraint(name,v,dom)

Create a new constraint in the model.

```
name :   String

v :   Variable

dom :   PSDDomain

returns : Constraint
```

5.22.5.12 Model.constraint(v,dom)

Create a new constraint in the model.

```
v :    Variable

dom :   PSDDomain

returns : Constraint
```

5.22.5.13 Model.constraint(name,shape,v,dom)

Create a new constraint in the model.

```
name :   String

shape :   Set

v :    Variable

dom :   Domain

returns : Constraint
```

5.22.5.14 Model.constraint(shape,v,dom)

Create a new constraint in the model.

```
shape :   Set

v :    Variable

dom :   Domain

returns : Constraint
```

5.22.5.15 Model.constraint(name,v,dom)

Create a new constraint in the model.

```
name :   String

v :    Variable

dom :   Domain

returns : Constraint
```

5.22.5.16   Model.constraint(v,dom)

Create a new constraint in the model.

```
v  :    Variable

dom :   Domain

returns : Constraint
```

5.22.5.17   Model.constraint(name,shape,v,dom)

Create a new constraint in the model.

```
name  :   String

shape  :    Set

v  :    Variable

dom  :    RangeDomain

returns : Constraint
```

5.22.5.18   Model.constraint(shape,v,dom)

Create a new constraint in the model.

```
shape  :    Set

v  :    Variable

dom  :    RangeDomain

returns : Constraint
```

5.22.5.19   Model.constraint(name,v,dom)

Create a new constraint in the model.

```
name  :   String

v  :    Variable

dom  :    RangeDomain

returns : Constraint
```

5.22.5.20   Model.constraint(v,dom)

Create a new constraint in the model.

```
v :   Variable
```

```
dom :   RangeDomain
```

```
returns : Constraint
```

5.22.6   Model.primalObjValue()

Member of `Model`.
```
Model.primalObjValue()
```

5.22.6.1   Model.primalObjValue()

Get the primal objective value.

Throws exceptions :

- SolutionError

```
returns : double
```

5.22.7   Model.getSolverDoubleInfo()

Member of `Model`.
```
Model.getSolverDoubleInfo(String name)
```

5.22.7.1   Model.getSolverDoubleInfo(name)

Return the floating point value for the solution information selected by the input parameter. The list of available parameters can be found in Section (5.43.1).

If the parameter name does not correspond to any information available as a floating point number, a runtime exception is thrown.

```
name :   String
```
     The parameter name.

```
returns : double
```

5.22.8 Model.sparseVariable()

Member of Model.

```
Model.sparseVariable(String name,int size,Domain dom)
Model.sparseVariable(String name,int size,RangeDomain dom)
Model.sparseVariable(String name,int size,Domain dom,IntegerDomain idom)
Model.sparseVariable(String name,int size,RangeDomain dom,IntegerDomain idom)
Model.sparseVariable(String name,Set shp,Domain dom)
Model.sparseVariable(String name,Set shp,RangeDomain dom)
Model.sparseVariable(String name,Set shp,Domain dom,IntegerDomain idom)
Model.sparseVariable(String name,Set shp,RangeDomain dom,IntegerDomain idom)
Model.sparseVariable(int size,Domain dom)
Model.sparseVariable(int size,RangeDomain dom)
Model.sparseVariable(int size,Domain dom,IntegerDomain idom)
Model.sparseVariable(int size,RangeDomain dom,IntegerDomain idom)
Model.sparseVariable(Set shp,Domain dom)
Model.sparseVariable(Set shp,RangeDomain dom)
Model.sparseVariable(Set shp,Domain dom,IntegerDomain idom)
Model.sparseVariable(Set shp,RangeDomain dom,IntegerDomain idom)
Model.sparseVariable(String name,Domain dom)
Model.sparseVariable(String name,RangeDomain dom)
Model.sparseVariable(String name,Domain dom,IntegerDomain idom)
Model.sparseVariable(String name,RangeDomain dom,IntegerDomain idom)
Model.sparseVariable(Domain dom)
Model.sparseVariable(RangeDomain dom)
Model.sparseVariable(Domain dom,IntegerDomain idom)
Model.sparseVariable(RangeDomain dom,IntegerDomain idom)
```

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

dom : RangeDomain

> Defines the domain of the variable.

idom : IntegerDomain

> If not null it indicates that the variable is integer constrained.

name : String

> Name of the variable. This must be unique among all variables in the model. The value `null` is allowed instead of a unique name.

shp : Set

> Defines the shape of the variable.

size : int

> Size of the variable. The variable becomes a one-dimensional vector of the given size.

returns : Variable

    A new variable object.

5.22.8.1 Model.sparseVariable(name,size,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :  String

size :  int

dom :  Domain
```

returns : `Variable`

5.22.8.2 Model.sparseVariable(name,size,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :  String

size :  int

dom :  RangeDomain
```

returns : `Variable`

5.22.8.3 Model.sparseVariable(name,size,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This

means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :  String

size :  int

dom :   Domain

idom :  IntegerDomain

returns : Variable
```

5.22.8.4   Model.sparseVariable(name,size,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :  String

size :  int

dom :   RangeDomain

idom :  IntegerDomain

returns : Variable
```

5.22.8.5   Model.sparseVariable(name,shp,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :  String
```

```
shp :  Set

dom :  Domain
```

returns : `Variable`

5.22.8.6 Model.sparseVariable(name,shp,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :  String

shp :  Set

dom :  RangeDomain
```

returns : `Variable`

5.22.8.7 Model.sparseVariable(name,shp,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :  String

shp :  Set

dom :  Domain

idom :  IntegerDomain
```

returns : `Variable`

5.22.8.8 Model.sparseVariable(name,shp,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name  :   String

shp  :   Set

dom  :   RangeDomain

idom  :   IntegerDomain

returns : Variable
```

5.22.8.9 Model.sparseVariable(size,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
size  :   int

dom  :   Domain

returns : Variable
```

5.22.8.10 Model.sparseVariable(size,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
size :  int

dom :   RangeDomain

returns : Variable
```

5.22.8.11   Model.sparseVariable(size,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
size :  int

dom :   Domain

idom :  IntegerDomain

returns : Variable
```

5.22.8.12   Model.sparseVariable(size,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
size :  int

dom :   RangeDomain

idom :  IntegerDomain

returns : Variable
```

5.22.8.13   Model.sparseVariable(shp,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
 shp :   Set

 dom :   Domain

 returns : Variable
```

5.22.8.14   Model.sparseVariable(shp,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
 shp :   Set

 dom :   RangeDomain

 returns : Variable
```

5.22.8.15   Model.sparseVariable(shp,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
 shp :   Set

 dom :   Domain
```

```
idom :   IntegerDomain
```

```
returns : Variable
```

5.22.8.16   Model.sparseVariable(shp,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
shp :   Set
```

```
dom :   RangeDomain
```

```
idom :   IntegerDomain
```

```
returns : Variable
```

5.22.8.17   Model.sparseVariable(name,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :   String
```

```
dom :   Domain
```

```
returns : Variable
```

5.22.8.18   Model.sparseVariable(name,dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :   String

dom :   RangeDomain

returns : Variable
```

5.22.8.19   Model.sparseVariable(name,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :   String

dom :   Domain

idom :   IntegerDomain

returns : Variable
```

5.22.8.20   Model.sparseVariable(name,dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
name :   String

dom :   RangeDomain

idom :   IntegerDomain

returns : Variable
```

5.22.8.21 Model.sparseVariable(dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
dom :   Domain
```

```
returns : Variable
```

5.22.8.22 Model.sparseVariable(dom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
dom :   RangeDomain
```

```
returns : Variable
```

5.22.8.23 Model.sparseVariable(dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
dom :   Domain
```

```
idom :   IntegerDomain
```

```
returns : Variable
```

5.22.8.24   Model.sparseVariable(dom,idom)

Create a new sparse variable in the model.

The elements in the sparse variable are not instantiated until they are used in a constraint or objective, i.e. the memory used to represent the variable is proportional to the number og elements used. This means that the variable can be declared to have maximum theoretical size, as long as only a subset of elements are used.

Note that using sparse variables is slower than dense variables so they should only be used when the variable is in fact sparse since.

```
dom :   RangeDomain

idom :   IntegerDomain

returns : Variable
```

5.22.9   Model.getSolverIntInfo()

Member of Model.

```
Model.getSolverIntInfo(String name)
```

5.22.9.1   Model.getSolverIntInfo(name)

Return the value integer for the solution information selected by the input parameter. The list of the available integer parameters can be found in Section (5.43.2).

If the parameter name does not correspond to any information available as an integer number, a runtime exception is thrown.

```
name :   String
```
   The parameter name.

```
returns : int
```

5.22.10   Model.objective()

Member of Model.

```
Model.objective(String name,ObjectiveSense sense,Expression expr)
Model.objective(String name,ObjectiveSense sense,Variable v)
Model.objective(String name,ObjectiveSense sense,double c)
Model.objective(String name,double c)
Model.objective(ObjectiveSense sense,Expression expr)
Model.objective(ObjectiveSense sense,Variable v)
Model.objective(ObjectiveSense sense,double c)
Model.objective(double c)
```

Replace the objective expression.

expr : Expression

    The objective expression. This must be an expression containing exactly one row.

name : String

    Name of the obective; this may be any string, and its has no function except when writing the problem to an external file formal.

sense : ObjectiveSense

    The objective sense; defines whether the objective must be minimized or maximized.

v : Variable

    The objective variable. This must be a scaler variable.

5.22.10.1   Model.objective(name,sense,expr)

Replace the objective expression.

```
name :  String
```

```
sense :  ObjectiveSense
```

```
expr :  Expression
```

5.22.10.2   Model.objective(name,sense,v)

Replace the objective expression.

```
name :  String
```

```
sense :  ObjectiveSense
```

```
v :  Variable
```

5.22.10.3   Model.objective(name,sense,c)

Replace the objective expression.

```
name :  String
```

```
sense :  ObjectiveSense
```

```
c :  double
```

5.22.10.4   Model.objective(name,c)

Replace the objective expression.

```
name :  String
```

```
c :  double
```

5.22.10.5   Model.objective(sense,expr)

Replace the objective expression.

```
sense :  ObjectiveSense
```

```
expr :  Expression
```

5.22.10.6   Model.objective(sense,v)

Replace the objective expression.

```
sense :  ObjectiveSense
```

```
v :  Variable
```

5.22.10.7   Model.objective(sense,c)

Replace the objective expression.

```
sense :  ObjectiveSense
```

```
c :  double
```

5.22.10.8   Model.objective(c)

Replace the objective expression.

```
c :  double
```

5.22.11   Model.setLogHandler()

Member of Model.
```
Model.setLogHandler(java.io.Writer h)
```

5.22.11.1 Model.setLogHandler(h)

The solver log information will be sent to the stream handler.

 h : java.io.Writer
   The log handler object or `null`.

5.22.12 Model.getDualSolutionStatus()

Member of `Model`.

`Model.getDualSolutionStatus()`

5.22.12.1 Model.getDualSolutionStatus()

Return the status of the dual solution.

 returns : `SolutionStatus`

5.22.13 Model.setSolverParam()

Member of `Model`.

`Model.setSolverParam(String name,String strval)`
`Model.setSolverParam(String name,int intval)`
`Model.setSolverParam(String name,double floatval)`

Set a solver parameter

Solver parameter values can be either symbolic values, integers or doubles, depending on the parameter. The value is automatically converted to a suitable type, or, if this fails, an exception will be thrown. For example, if the parameter accepts a double value and is give a string, the string will be parsed to produce a double.

See 5.42 for a listing of all parameter settings.

 floatval : double
   A float value to assign to the parameter.

 intval : int
   An integer value to assign to the parameter.

 name : String
   Name of the parameter to set

 strval : String
   A string value to assign to the parameter.

5.22.13.1   Model.setSolverParam(name,strval)

Set a solver parameter as a string value. If the parametere accepts an integer or a double, the string will be converted to the relevant type. Otherwise, the value must be a symbolic value recognized by the parameter.

See 5.42 for a listing of all parameter settings.

```
name :  String
```
     Name of the parameter to set

```
strval :  String
```
     A string value to assign to the parameter.

5.22.13.2   Model.setSolverParam(name,intval)

Set an integer or floating point parameter.

If the parameter expects a double value, the value is converted. If the parameter expects a symbolic value, an exception will be thrown.

See 5.42 for a listing of all parameter settings.

```
name :  String
```
     Name of the parameter to set

```
intval :  int
```
     An integer value to assign to the parameter.

5.22.13.3   Model.setSolverParam(name,floatval)

Set a floating point parameter.

If the parameter expects an integer or a symbolic value, an exception will be thrown.

See 5.42 for a listing of all parameter settings.

```
name :  String
```
     Name of the parameter to set

```
floatval :  double
```
     A float value to assign to the parameter.

5.22.14   Model.getSolverLIntInfo()

Member of Model.

```
Model.getSolverLIntInfo(String name)
```

5.22.14.1 Model.getSolverLIntInfo(name)

Return the value for the long integer solution information selected by the input parameter. The list of the available long integer parameters can be found in Section (5.43.3).

If the parameter name does not correspond to any information available as a long integer number, a runtime exception is thrown.

```
name :  String
```
    The parameter name.

```
returns : long
```

5.22.15 Model.getVariable()

Member of `Model`.
```
Model.getVariable(String name)
```

5.22.15.1 Model.getVariable(name)

Get the variable corresponding to the given name

```
name :  String
```
    The variable name.

```
returns : Variable
```
    The variable object with the given name.

5.22.16 Model.variable()

Member of `Model`.
```
Model.variable(String name,int size,Domain dom)
Model.variable(String name,int size,RangeDomain dom)
Model.variable(String name,int size,Domain dom,IntegerDomain idom)
Model.variable(String name,int size,RangeDomain dom,IntegerDomain idom)
Model.variable(String name,Set shp,Domain dom)
Model.variable(String name,Set shp,RangeDomain dom)
Model.variable(String name,Set shp,Domain dom,IntegerDomain idom)
Model.variable(String name,Set shp,RangeDomain dom,IntegerDomain idom)
Model.variable(int size,Domain dom)
Model.variable(int size,RangeDomain dom)
Model.variable(int size,Domain dom,IntegerDomain idom)
Model.variable(int size,RangeDomain dom,IntegerDomain idom)
Model.variable(Set shp,Domain dom)
Model.variable(Set shp,RangeDomain dom)
Model.variable(Set shp,Domain dom,IntegerDomain idom)
Model.variable(Set shp,RangeDomain dom,IntegerDomain idom)
```

```
Model.variable(String name,Domain dom)
Model.variable(String name,RangeDomain dom)
Model.variable(String name,Domain dom,IntegerDomain idom)
Model.variable(String name,RangeDomain dom,IntegerDomain idom)
Model.variable(Domain dom)
Model.variable(RangeDomain dom)
Model.variable(Domain dom,IntegerDomain idom)
Model.variable(RangeDomain dom,IntegerDomain idom)
Model.variable(String name,Set shp,PSDDomain dom)
Model.variable(String name,int n,PSDDomain dom)
Model.variable(String name,int n,int m,PSDDomain dom)
Model.variable(String name,PSDDomain dom)
Model.variable(int n,PSDDomain dom)
Model.variable(int n,int m,PSDDomain dom)
Model.variable(PSDDomain dom)
```

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a Domain or a RangeDomain.

dom : PSDDomain

> Defines the domain of the variable. The shape and the domain must match: The domain must either be scalable, e.g. `Domain.equalsTo(0.0)`, or the size and shape must be matched by the shape defined by either `shape` or `size`.

idom : IntegerDomain

> If not null it indicates that the variable is integer constrained.

name : String

> Name of the variable. This must be unique among all variables in the model. The value `null` is allowed instead of a unique name.

shp : Set

> Defines the shape of the variable.

size : int

> Size of the variable. The variable becomes a one-dimensional vector of the given size.

5.22.16.1   Model.variable(name,size,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a Domain or a RangeDomain.

```
name :  String
```

```
size :  int
```

```
dom :  Domain
```

returns : Variable

5.22.16.2 Model.variable(name,size,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a Domain or a RangeDomain.

```
name :  String
```

```
size :  int
```

```
dom :  RangeDomain
```

returns : Variable

5.22.16.3 Model.variable(name,size,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a Domain or a RangeDomain.

```
name :  String
```

```
size :  int
```

```
dom :   Domain
```

```
idom :   IntegerDomain
```

returns : Variable

5.22.16.4   Model.variable(name,size,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a Domain or a RangeDomain.

```
name :   String
```

```
size :   int
```

```
dom :   RangeDomain
```

```
idom :   IntegerDomain
```

returns : Variable

5.22.16.5   Model.variable(name,shp,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a Domain or a RangeDomain.

```
name :   String
```

```
shp :   Set
```

```
dom :   Domain
```

returns : Variable

5.22.16.6 Model.variable(name,shp,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a Domain or a RangeDomain.

```
name :   String

shp :    Set

dom :    RangeDomain

returns : Variable
```

5.22.16.7 Model.variable(name,shp,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a Domain or a RangeDomain.

```
name :   String

shp :    Set

dom :    Domain

idom :   IntegerDomain

returns : Variable
```

5.22.16.8 Model.variable(name,shp,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a Domain or a RangeDomain.

```
name :  String

shp :   Set

dom :   RangeDomain

idom :   IntegerDomain

returns : Variable
```

5.22.16.9   Model.variable(size,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a Domain or a RangeDomain.

```
size :   int

dom :   Domain

returns : Variable
```

5.22.16.10   Model.variable(size,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a Domain or a RangeDomain.

```
size :   int

dom :   RangeDomain

returns : Variable
```

5.22.16.11   Model.variable(size,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a Domain or a RangeDomain.

```
size :   int
```

```
dom :   Domain
```

```
idom :   IntegerDomain
```

returns : Variable

5.22.16.12   Model.variable(size,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a Domain or a RangeDomain.

```
size :   int
```

```
dom :   RangeDomain
```

```
idom :   IntegerDomain
```

returns : Variable

5.22.16.13   Model.variable(shp,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

 shp :   `Set`

 dom :   `Domain`

 returns : `Variable`

5.22.16.14   Model.variable(shp,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

 `variable(name, shp, dom, idom)`

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

 shp :   `Set`

 dom :   `RangeDomain`

 returns : `Variable`

5.22.16.15   Model.variable(shp,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

 `variable(name, shp, dom, idom)`

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

 shp :   `Set`

 dom :   `Domain`

 idom :   `IntegerDomain`

 returns : `Variable`

5.22.16.16 Model.variable(shp,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
shp :   Set

dom :   RangeDomain

idom :   IntegerDomain

returns : Variable
```

5.22.16.17 Model.variable(name,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
name :   String

dom :   Domain

returns : Variable
```

5.22.16.18 Model.variable(name,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
name :  String
```

```
dom :   RangeDomain
```

returns : `Variable`

5.22.16.19   Model.variable(name,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
name :  String
```

```
dom :   Domain
```

```
idom :  IntegerDomain
```

returns : `Variable`

5.22.16.20   Model.variable(name,dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
name :  String
```

```
dom :   RangeDomain
```

```
idom :  IntegerDomain
```

returns : `Variable`

5.22.16.21   Model.variable(dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
dom :   Domain
```

```
returns : Variable
```

5.22.16.22   Model.variable(dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
dom :   RangeDomain
```

```
returns : Variable
```

5.22.16.23   Model.variable(dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
dom :   Domain
```

```
idom :   IntegerDomain
```

```
returns : Variable
```

5.22.16.24   Model.variable(dom,idom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
dom  :   RangeDomain

idom :   IntegerDomain
```

returns : `Variable`

5.22.16.25   Model.variable(name,shp,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
name :   String

shp  :   Set

dom  :   PSDDomain
```

returns : `Variable`

5.22.16.26   Model.variable(name,n,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
name :  String

n :  int

dom :  PSDDomain

returns : Variable
```

5.22.16.27   Model.variable(name,n,m,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a Domain or a RangeDomain.

```
name :  String

n :  int

m :  int

dom :  PSDDomain

returns : Variable
```

5.22.16.28   Model.variable(name,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of `name`, `shp` and `idom` can be left out, and dom can be either a Domain or a RangeDomain.

```
name :  String

dom :  PSDDomain

returns : Variable
```

5.22.16.29   Model.variable(n,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
n  :   int
```

```
dom  :   PSDDomain
```

returns : `Variable`

5.22.16.30   Model.variable(n,m,dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
n  :   int
```

```
m  :   int
```

```
dom  :   PSDDomain
```

returns : `Variable`

5.22.16.31   Model.variable(dom)

Create a new variable in the model.

The shape and the domain of the variable must match.

There are a long list of overloaded methods for variable creation, but they are all variations over the same method:

```
variable(name, shp, dom, idom)
```

where any or all of **name**, **shp** and **idom** can be left out, and dom can be either a `Domain` or a `RangeDomain`.

```
dom  :  PSDDomain
```

```
returns : Variable
```

5.22.17 Model.solve()

Member of `Model`.

```
Model.solve()
```

5.22.17.1 Model.solve()

This calls MOSEK solver to solve the problem defined in the model. Upon return the solution status (see `Model.getPrimalSolutionStatus`) and `Model.getDualSolutionStatus`) and values will be available.

Depending on the solution status, various values may be defined:

- If the model is primal-dual feasible, or nearly so, and the solver found a solution, the solution values can be accessed through the `Variable` and `Constraint` objects in the model.

  For integer problems only the primal solution is defined, while for continuous problems both primal and dual solutions are available.

- The model is primal or dual infeaible. In this case *only* the primal *or* the dual solution is defined, depending on the solution status.

- The solver ran into problems and did not find anything useful. In this case the solution values may be garbage.

Separate solution status values are defined for the primal and the dual solutions. These can be obtained with `Model.getPrimalSolutionStatus` and `Model.getDualSolutionStatus`.

By default, trying to fetch a non-optimal solution using `Variable.level` or `Variable.dual` will cause an exception. To fetch certificates or other values it is necessary to change the accepted solution flag; see `Model.acceptedSolutionStatus`.

Throws exceptions :

- `OptimizeError`

5.22.18 Model.writeTask()

Member of `Model`.

```
Model.writeTask(String filename)
```

5.22.18.1   Model.writeTask(filename)

The file extension is used to determine the file format.

```
filename :   String
```
    Name of the file to write.

5.22.19   `Model.dualObjValue()`

Member of `Model`.
```
Model.dualObjValue()
```

5.22.19.1   Model.dualObjValue()

Get the dual objective value.

  Throws exceptions :

- `SolutionError`

returns : `double`

5.23   `fusion.ModelConstraint`

Base class for all constraints that directly corresponds to a block of constraints in the underlying task, i.e. all objects created from `Model.constraint`.

  Base class
    `Constraint`

  Known direct descendants
    `LinearConstraint`, `PSDConstraint`, `RangedConstraint`, `ConicConstraint`

  Methods

    `ModelConstraint.slice`
    `ModelConstraint.toString`

  Inherited methods

    `Constraint.add(Expression expr)`
        Add an expression to the constraint expression.
    `Constraint.add(Variable v)`
        Add an expression to the constraint expression.

Constraint.dual()
> Get the dual solution value of the variable.

Constraint.dual(int[] firstidx,int[] lastidx)
> Return a slice of the dual solution.

Constraint.dual(int firstidx,int lastidx)
> Return a slice of the dual solution.

Constraint.get_model()
> Get the original model object.

Constraint.get_nd()
> Get the number of dimensions of the constraint.

Constraint.index(int[] idx)
> Get a single element from a one-dimensional constraint.

Constraint.index(int idx)
> Get a single element from a one-dimensional constraint.

Constraint.level(int index)
> Return a single value of the primal solution.

Constraint.level(int firstidx,int lastidx)
> Return a slice of the primal solution.

Constraint.level()
> Get the primal solution value of the variable.

Constraint.level(int[] firstidx,int[] lastidx)
> Return a slice of the primal solution.

Constraint.size()
> Get the total number of elements in the constraint.

5.23.1   ModelConstraint.slice()

Member of ModelConstraint.

```
ModelConstraint.slice(int first,int last)
ModelConstraint.slice(int[] first,int[] last)
```

5.23.1.1   ModelConstraint.slice(first,last)

Get a slice of the constraint.

Assuming that the shape of the constraint is one-dimensional, this function creates a constraint slice of (last-first+1) elements, which acts as an interface to a sub-vector of this constraint.

first : int
> Index of the first element in the slice.

`last :  int`

  Index if the last element in the slice.

returns : `Constraint`

  A new constraint object representing a slice of this object.

5.23.1.2 ModelConstraint.slice(first,last)

Get a multi-dimensional slice of the constraint.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the constraint.

`first :  int[]`

  Array of start elements in the slice.

`last :  int[]`

  Array of end element in the slice.

returns : `Constraint`

  A new constraint object representing a slice of this object.

5.23.2 `ModelConstraint.toString()`

Member of `ModelConstraint`.

`ModelConstraint.toString()`

5.23.2.1 ModelConstraint.toString()

returns : `String`

5.24 `fusion.ModelVariable`

Base class for all variables that directly corresponds to a block of variables in the underlying task, i.e. all objects created from `Model.variable`.

Base class

  `Variable`

Known direct descendants

  `ConicVariable`, `RangedVariable`, `LinearVariable`, `PSDVariable`

Methods

`ModelVariable.slice`

`ModelVariable.toString`
> Create a string-representation of the variable.

Inherited methods

`Variable.antidiag(int index)`
> Return an anti-diagonal of a square matrix.

`Variable.antidiag()`
> Return the anti-diagonal of a square matrix

`Variable.asExpr()`
> Create an expression corresponding to the variable object.

`Variable.compress()`
> Reshape a variable object by removing all dimensions of size 1.

`Variable.diag()`
> Return the diagonal of a square matrix

`Variable.diag(int index)`
> Return a diagonal of a square matrix.

`Variable.dual(int[] firstidx,int[] lastidx)`
> Return a slice of the dual solution.

`Variable.dual(int firstidx,int lastidx)`
> Return a slice of the dual solution.

`Variable.dual()`
> Get the dual solution value of the variable.

`Variable.index(int[] idx)`
> Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1)`
> Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1,int i2)`
> Get a single element from a one-dimensional variable.

`Variable.index(int i0)`
> Get a single element from a one-dimensional variable.

`Variable.index_flat(long i)`
> Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`
> Return a single value of the primal solution.

`Variable.level()`
> Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`

>   Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`

>   Return a slice of the primal solution.

`Variable.pick(int[] idxs)`

>   Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`

>   Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`

>   Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`

>   Get the number of elements in the variable.

`Variable.transpose()`

>   Transpose a vector or matrix variable

5.24.1   `ModelVariable.slice()`

Member of `ModelVariable`.

```
ModelVariable.slice(int first,int last)
ModelVariable.slice(int[] first,int[] last)
```

5.24.1.1   ModelVariable.slice(first,last)

Get a slice of the variable.

Assuming that the shape of the variable is one-dimensional, this function creates a variable slice of (last-first+1) elements, which acts as an interface to a sub-vector of this variable.

`first :  int`

>   Index of the first element in the slice.

`last :  int`

>   Index if the last element plus one in the slice.

returns : `Variable`

>   A new variable object representing a slice of this object.

5.24.1.2   ModelVariable.slice(first,last)

Get a multi-dimensional slice of the variable.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the variable.

`first : int[]`

>   Array of start elements in the slice.

`last : int[]`

>   Array of end elements plus one in the slice.

returns : `Variable`

>   A new variable object representing a slice of this object.

5.24.2   `ModelVariable.toString()`

Member of `ModelVariable`.

`ModelVariable.toString()`

5.24.2.1   ModelVariable.toString()

Create a string-representation of the variable.

returns : `String`

>   A string representing the variable.

5.25   `fusion.NDSet`

N-dimensional integer set. ;

Base class

>   `Set`

Known direct descendants

>   `ProductSet`

Constructors

>   `NDSet(int[] startx,int[] stopx)`
>   >   Construct an N-dimensional set with given start and stop values.

NDSet(int[] sizes)

> Construct an N-dimensional set, where the size of each dimension if given.

NDSet(int size0,int size1)

> Construct a 2-dimensional set, where the size of each dimension if given.

NDSet(int size0,int size1,int size2)

> Construct an 3-dimensional set, where the size of each dimension if given.

Methods

NDSet.dim

> Return the size of the given dimension.

NDSet.getname

NDSet.indexToString

NDSet.slice

NDSet.stride

> Return the stride size in the given dimension.

Inherited methods

Set.compare(Set other)

> Compare two sets and return true if they have the same shape and size.

Set.getSize()

> Total number of elements in the set.

Set.idxtokey(long idx)

> Convert a linear index to a N-dimensional key.

Set.realnd()

> Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

Set.toString()

> Return a string representation of the set.

5.25.1   NDSet.NDSet()

Member of NDSet.

```
NDSet.NDSet(int[] startx,int[] stopx)
NDSet.NDSet(int[] sizes)
NDSet.NDSet(int size0,int size1)
NDSet.NDSet(int size0,int size1,int size2)
```

5.25.1.1 NDSet.NDSet(startx,stopx)

Construct an N-dimensional set with given start and stop values.

The arrays of start and stop indexes must have the same length. Note that as for integer sets both indexes are incluse; i.e. a set with start=1 and stop=10 means 1,2,3,4,5,6,7,8,9,10

`startx : int[]`

    Array of start indexes for each dimension.

`stopx : int[]`

    Array of end indexes for each dimension.

5.25.1.2 NDSet.NDSet(sizes)

Construct an N-dimensional set, where the size of each dimension if given.

`sizes : int[]`

    Array of dimension sizes.

5.25.1.3 NDSet.NDSet(size0,size1)

Construct a 2-dimensional set, where the size of each dimension if given.

`size0 : int`

    Size of first dimension.

`size1 : int`

    Size of second dimension.

5.25.1.4 NDSet.NDSet(size0,size1,size2)

Construct an 3-dimensional set, where the size of each dimension if given.

`size0 : int`

    Size of first dimension.

`size1 : int`

    Size of second dimension.

`size2 : int`

5.25.2 NDSet.stride()

Member of NDSet.

```
NDSet.stride(int i)
```

5.25.2.1 NDSet.stride(i)

Return the stride size in the given dimension.

i : int

> Dimension index.

returns : long

> The stride size in the requested dimension.

5.25.3 NDSet.getname()

Member of NDSet.

```
NDSet.getname(long key)
NDSet.getname(int[] key)
```

5.25.3.1 NDSet.getname(key)

Return a string representing the index.

key : long

> A linear index.

returns : String

> Get a string representing the item identified by the key.

5.25.3.2 NDSet.getname(key)

Return a string representing the index.

key : int[]

> An N-dimensional index.

returns : String

> Get a string representing the item identified by the key.

5.25.4   `NDSet.slice()`

Member of NDSet.

NDSet.slice`(int first,int last)`
NDSet.slice`(int[] first,int[] last)`

5.25.4.1   NDSet.slice(first,last)

Create a set object representing a slice of this set.

`first :   int`

Last index in the range.

`last :   int`

returns : Set

A new Set object representing the slice.

5.25.4.2   NDSet.slice(first,last)

Create a set object representing a slice of this set.

`first :   int[]`

Last index in each dimension in the range.

`last :   int[]`

returns : Set

A new Set object representing the slice.

5.25.5   `NDSet.indexToString()`

Member of NDSet.

NDSet.indexToString`(long index)`

5.25.5.1   NDSet.indexToString(index)

`index :   long`

returns : `String`

5.25.6   `NDSet.dim()`

Member of NDSet.

NDSet.dim`(int i)`

5.25.6.1   NDSet.dim(i)

Return the size of the given dimension.

i :   int
    Dimension index.

returns : int
    The size of the requested dimension.

5.26   `fusion.PSDConstraint`

This class represents a semidefinite conic constraint of the form

$$Ax - b \preceq 0$$

i.e. $Ax - b$ must be positive semidefinite

Base class
    ModelConstraint

Inherited methods

    Constraint.add(Expression expr)
        Add an expression to the constraint expression.
    Constraint.add(Variable v)
        Add an expression to the constraint expression.
    Constraint.dual()
        Get the dual solution value of the variable.
    Constraint.dual(int[] firstidx,int[] lastidx)
        Return a slice of the dual solution.
    Constraint.dual(int firstidx,int lastidx)
        Return a slice of the dual solution.
    Constraint.get_model()
        Get the original model object.
    Constraint.get_nd()
        Get the number of dimensions of the constraint.
    Constraint.index(int[] idx)
        Get a single element from a one-dimensional constraint.
    Constraint.index(int idx)
        Get a single element from a one-dimensional constraint.

Constraint.level(int index)
> Return a single value of the primal solution.

Constraint.level(int firstidx,int lastidx)
> Return a slice of the primal solution.

Constraint.level()
> Get the primal solution value of the variable.

Constraint.level(int[] firstidx,int[] lastidx)
> Return a slice of the primal solution.

Constraint.size()
> Get the total number of elements in the constraint.

ModelConstraint.slice(int first,int last)
> Get a slice of the constraint.

ModelConstraint.slice(int[] first,int[] last)
> Get a multi-dimensional slice of the constraint.

ModelConstraint.toString()

5.27 `fusion.PSDDomain`

5.28 `fusion.PSDVariable`

This class represents a positive semidefinite variable.

Base class
> ModelVariable

Inherited methods

Variable.antidiag(int index)
> Return an anti-diagonal of a square matrix.

Variable.antidiag()
> Return the anti-diagonal of a square matrix

Variable.asExpr()
> Create an expression corresponding to the variable object.

Variable.compress()
> Reshape a variable object by removing all dimensions of size 1.

Variable.diag()
> Return the diagonal of a square matrix

Variable.diag(int index)
> Return a diagonal of a square matrix.

`Variable.dual(int[] firstidx,int[] lastidx)`
    Return a slice of the dual solution.
`Variable.dual(int firstidx,int lastidx)`
    Return a slice of the dual solution.
`Variable.dual()`
    Get the dual solution value of the variable.
`Variable.index(int[] idx)`
    Get a single element from a one-dimensional variable.
`Variable.index(int i0,int i1)`
    Get a single element from a one-dimensional variable.
`Variable.index(int i0,int i1,int i2)`
    Get a single element from a one-dimensional variable.
`Variable.index(int i0)`
    Get a single element from a one-dimensional variable.
`Variable.index_flat(long i)`
    Index into the variable as if it was a one-dimensional object.
`Variable.level(int index)`
    Return a single value of the primal solution.
`Variable.level()`
    Get the primal solution value of the variable.
`Variable.level(int[] firstidx,int[] lastidx)`
    Return a slice of the primal solution.
`Variable.level(int firstidx,int lastidx)`
    Return a slice of the primal solution.
`Variable.pick(int[] idxs)`
    Create a vector-variable by picking a list of indexes from this variable.
`Variable.pick(int[][] midxs)`
    Create a matrix-variable by picking a list of indexes from this variable.
`Variable.pick_flat(long[] indexes)`
    Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.
`Variable.size()`
    Get the number of elements in the variable.
`ModelVariable.slice(int[] first,int[] last)`
    Get a multi-dimensional slice of the variable.
`ModelVariable.slice(int first,int last)`
    Create a variable object representing a slice of the variable.
`ModelVariable.toString()`
    Create a string-representation of the variable.
`Variable.transpose()`
    Transpose a vector or matrix variable

5.29 fusion.ProductSet

One-dimensional set defined as a range of inregers.

Base class
> NDSet

Constructors

> ProductSet(Set[] ss)
>> Construct set from a list of other sets.

Methods

> ProductSet.indexToString

Inherited methods

> Set.compare(Set other)
>> Compare two sets and return true if they have the same shape and size.

> NDSet.dim(int i)
>> Return the size of the given dimension.

> Set.getSize()
>> Total number of elements in the set.

> NDSet.getname(int[] key)
>> Return a string representing the index.

> NDSet.getname(long key)
>> Return a string representing the index.

> Set.idxtokey(long idx)
>> Convert a linear index to a N-dimensional key.

> Set.realnd()
>> Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

> NDSet.slice(int first,int last)
>> Create a set object representing a slice of this set.

> NDSet.slice(int[] first,int[] last)
>> Create a set object representing a slice of this set.

> NDSet.stride(int i)
>> Return the stride size in the given dimension.

> Set.toString()
>> Return a string representation of the set.

5.29.1 ProductSet.ProductSet()

Member of ProductSet.

```
ProductSet.ProductSet(Set[] ss)
```

5.29.1.1 ProductSet.ProductSet(ss)

Construct set from a list of other sets.

```
ss :   Set[]
```

5.29.2 ProductSet.indexToString()

Member of ProductSet.

```
ProductSet.indexToString(long index)
```

5.29.2.1 ProductSet.indexToString(index)

```
index :   long
```

```
returns : String
```

5.30   fusion.RangeDomain

The RangeDomain object is never instantiated directly: Instead use the relevant methods in Domain.

5.31   fusion.RangedConstraint

Defines a ranged constraint.

Since this actually defines one constraint with two inequalities, there will be two dual values (slc and suc) corresponding to the lower and upper bounds. When asked for the dual solution, this constraint will return (y=slc-suc), but in some cases this is not enough (the individual dual constraints may be required for a certificate of infeasibility). The methods RangedConstraint.lowerBoundCon and RangedConstraint.upperBoundCon returns Variable objects that interface to the lower and upper bounds respectively.

Base class

    ModelConstraint

Methods

RangedConstraint.lowerBoundCon
    Get a constraint object corresponding to the lower bound of the ranged constraint.

RangedConstraint.upperBoundCon
    Get a constraint object corresponding to the upper bound of the ranged constraint.

Inherited methods

Constraint.add(Expression expr)
    Add an expression to the constraint expression.

Constraint.add(Variable v)
    Add an expression to the constraint expression.

Constraint.dual()
    Get the dual solution value of the variable.

Constraint.dual(int[] firstidx,int[] lastidx)
    Return a slice of the dual solution.

Constraint.dual(int firstidx,int lastidx)
    Return a slice of the dual solution.

Constraint.get_model()
    Get the original model object.

Constraint.get_nd()
    Get the number of dimensions of the constraint.

Constraint.index(int[] idx)
    Get a single element from a one-dimensional constraint.

Constraint.index(int idx)
    Get a single element from a one-dimensional constraint.

Constraint.level(int index)
    Return a single value of the primal solution.

Constraint.level(int firstidx,int lastidx)
    Return a slice of the primal solution.

Constraint.level()
    Get the primal solution value of the variable.

Constraint.level(int[] firstidx,int[] lastidx)
    Return a slice of the primal solution.

Constraint.size()
    Get the total number of elements in the constraint.

ModelConstraint.slice(int first,int last)
    Get a slice of the constraint.

ModelConstraint.slice(int[] first,int[] last)
    Get a multi-dimensional slice of the constraint.

ModelConstraint.toString()

5.31.1 `RangedConstraint.upperBoundCon()`

Member of `RangedConstraint`.

`RangedConstraint.upperBoundCon`()

5.31.1.1 **RangedConstraint.upperBoundCon()**

Get a constraint object corresponding to the upper bound of the ranged constraint.

returns : `Constraint`

> A new constraint object representing the upper bound of the constraint.

5.31.2 `RangedConstraint.lowerBoundCon()`

Member of `RangedConstraint`.

`RangedConstraint.lowerBoundCon`()

5.31.2.1 **RangedConstraint.lowerBoundCon()**

Get a constraint object corresponding to the lower bound of the ranged constraint.

returns : `Constraint`

> A new constraint object representing the lower bound of the constraint.

5.32 `fusion.RangedVariable`

Defines a ranged variable.

Since this actually defines one variable with two inequalities, there will be two dual variables (slx and sux) corresponding to the lower and upper bounds. When asked for the dual solution, this variable will return (y=slx-sux), but in some cases this is not enough (the individual dual variables may be required by e.g. a certificate). The methods `RangedVariable.lowerBoundVar` and `RangedVariable.upperBoundVar` returns Variable objects that interface to the lower and upper bounds respectively.

Base class

> `ModelVariable`

Known direct descendants

> `IntegerRangedVariable`

Methods

`RangedVariable.lowerBoundVar`
>   Get a variable object corresponding to the lower bound of the ranged variable.

`RangedVariable.upperBoundVar`
>   Get a variable object corresponding to the upper bound of the ranged variable.

Inherited methods

`Variable.antidiag(int index)`
>   Return an anti-diagonal of a square matrix.

`Variable.antidiag()`
>   Return the anti-diagonal of a square matrix

`Variable.asExpr()`
>   Create an expression corresponding to the variable object.

`Variable.compress()`
>   Reshape a variable object by removing all dimensions of size 1.

`Variable.diag()`
>   Return the diagonal of a square matrix

`Variable.diag(int index)`
>   Return a diagonal of a square matrix.

`Variable.dual(int[] firstidx,int[] lastidx)`
>   Return a slice of the dual solution.

`Variable.dual(int firstidx,int lastidx)`
>   Return a slice of the dual solution.

`Variable.dual()`
>   Get the dual solution value of the variable.

`Variable.index(int[] idx)`
>   Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1)`
>   Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1,int i2)`
>   Get a single element from a one-dimensional variable.

`Variable.index(int i0)`
>   Get a single element from a one-dimensional variable.

`Variable.index_flat(long i)`
>   Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`
>   Return a single value of the primal solution.

`Variable.level()`
>   Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`

Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`

Return a slice of the primal solution.

`Variable.pick(int[] idxs)`

Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`

Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`

Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`

Get the number of elements in the variable.

`ModelVariable.slice(int[] first,int[] last)`

Get a multi-dimensional slice of the variable.

`ModelVariable.slice(int first,int last)`

Create a variable object representing a slice of the variable.

`ModelVariable.toString()`

Create a string-representation of the variable.

`Variable.transpose()`

Transpose a vector or matrix variable

5.32.1   `RangedVariable.upperBoundVar()`

Member of `RangedVariable`.

`RangedVariable.upperBoundVar()`

5.32.1.1   **RangedVariable.upperBoundVar()**

Get a variable object corresponding to the upper bound of the ranged variable.

 returns : `Variable`

A variable object representing the upper bound of the variable.

5.32.2   `RangedVariable.lowerBoundVar()`

Member of `RangedVariable`.

`RangedVariable.lowerBoundVar()`

5.32.2.1   RangedVariable.lowerBoundVar()

Get a variable object corresponding to the lower bound of the ranged variable.

returns : `Variable`

 A variable object representing the lower bound of the variable.

5.33   `fusion.Set`

Base class shape specification objects.

Known direct descendants
 `BaseSet`, `NDSet`

Constructors

 `Set(int nd,long size)`
  The constructor method of Set

Methods

 `Set.compare`
  Compare two sets and return true if they have the same shape and size.

 `Set.dim`
  Return the size of the given dimension.

 `Set.getSize`
  Total number of elements in the set.

 `Set.getname`

 `Set.idxtokey`
  Convert a linear index to a N-dimensional key.

 `Set.indexToString`

 `Set.realnd`
  Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

 `Set.slice`

 `Set.stride`
  Return the stride size in the given dimension.

 `Set.toString`
  Return a string representation of the set.

Static Methods

 `Set.make`

5.33.1   `Set.Set()`

Member of `Set`.

`Set.Set(int nd,long size)`

5.33.1.1   **Set.Set(nd,size)**

The constructor method of Set

`nd :   int`
> Number of dimensions in the set.

`size :   long`
> Total number of elements in the set (the product of all dimensions).

5.33.2   `Set.dim()`

Member of `Set`.

`abstract Set.dim(int i)`

5.33.2.1   **abstract Set.dim(i)**

Return the size of the given dimension.

`i :   int`
> Dimension index.

`returns : int`
> The size of the requested dimension.

5.33.3   `Set.compare()`

Member of `Set`.

`Set.compare(Set other)`

5.33.3.1   **Set.compare(other)**

Note that the definition of "shape" only counts dimensions of size greater than 1. That is, compare will return true if a $N\times1$ is compared with a $1\times N$ set.

`other :   Set`

`returns : boolean`

5.33.4 `Set.slice()`

Member of Set.

```
 abstract Set.slice(int first,int last)
 abstract Set.slice(int[] first,int[] last)
```

5.33.4.1   abstract Set.slice(first,last)

Create a set object representing a slice of this set.

`first :  int`

> Last index in the range.

`last :  int`

`returns : Set`

> A new Set object representing the slice.

5.33.4.2   abstract Set.slice(first,last)

Create a set object representing a slice of this set.

`first :  int[]`

> Last index in each dimension in the range.

`last :  int[]`

`returns : Set`

> A new Set object representing the slice.

5.33.5 `Set.realnd()`

Member of Set.

```
 Set.realnd()
```

5.33.5.1   Set.realnd()

Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

`returns : int`

5.33.6 `Set.make()`

Member of Set.

```
static Set.make(String[] names)
static Set.make(int sz)
static Set.make(int s1,int s2)
static Set.make(int s1,int s2,int s3)
static Set.make(int[] sizes)
static Set.make(Set s1,Set s2)
static Set.make(Set[] ss)
```

5.33.6.1   static Set.make(names)

```
names :  String[]
```

```
returns : Set
```

5.33.6.2   static Set.make(sz)

```
sz :  int
```

```
returns : Set
```

5.33.6.3   static Set.make(s1,s2)

```
s1 :  int
```

```
s2 :  int
```

```
returns : Set
```

5.33.6.4   static Set.make(s1,s2,s3)

```
s1 :  int
```

```
s2 :  int
```

```
s3 :  int
```

```
returns : Set
```

5.33.6.5   static Set.make(sizes)

```
sizes :  int[]
```

```
returns : Set
```

5.33.6.6   static Set.make(s1,s2)

 s1 :   Set

 s2 :   Set

 returns : Set

5.33.6.7   static Set.make(ss)

 ss :   Set[]

 returns : Set

5.33.7   Set.getSize()

Member of Set.
 Set.getSize()

5.33.7.1   Set.getSize()

Total number of elements in the set.

 returns : long

5.33.8   Set.stride()

Member of Set.
 abstract Set.stride(int i)

5.33.8.1   abstract Set.stride(i)

Return the stride size in the given dimension.

 i :   int
      Dimension index.

 returns : long
      The stride size in the requested dimension.

5.33.9   Set.idxtokey()

Member of Set.
 Set.idxtokey(long idx)

5.33.9.1   Set.idxtokey(idx)

Convert a linear index to a N-dimensional key.

```
idx  :   long
```

    A linear index.

```
returns : int[]
```

    The N-dimensional key for the linear index.

5.33.10   Set.toString()

Member of Set.

```
Set.toString()
```

5.33.10.1   Set.toString()

Return a string representation of the set.

```
returns : String
```

    A string representation of the set.

5.33.11   Set.indexToString()

Member of Set.

```
abstract Set.indexToString(long index)
```

5.33.11.1   abstract Set.indexToString(index)

```
index  :   long
```

```
returns : String
```

5.33.12   Set.getname()

Member of Set.

```
abstract Set.getname(long key)
abstract Set.getname(int[] key)
```

5.33.12.1 abstract Set.getname(key)

Return a string representing the index.

`key : long`
> A linear index.

`returns : String`
> Get a string representing the item identified by the key.

5.33.12.2 abstract Set.getname(key)

Return a string representing the index.

`key : int[]`
> An N-dimensional index.

`returns : String`
> Get a string representing the item identified by the key.

5.34 fusion.SliceConstraint

An alias for a subset of constraints from a single ModelConstraint.

This class acts as a proxy for accessing a portion of a ModelConstraint. It is possible to access and modify the properties of the original variable using this alias. It does not access the Model directly, only through the original variable.

Base class
> Constraint

Known direct descendants
> BoundInterfaceConstraint

Methods
> SliceConstraint.size
> > Get the total number of elements in the constraint.
> SliceConstraint.slice

Inherited methods
> Constraint.add(Expression expr)
> > Add an expression to the constraint expression.

`Constraint.add(Variable v)`

> Add an expression to the constraint expression.

`Constraint.dual()`

> Get the dual solution value of the variable.

`Constraint.dual(int[] firstidx,int[] lastidx)`

> Return a slice of the dual solution.

`Constraint.dual(int firstidx,int lastidx)`

> Return a slice of the dual solution.

`Constraint.get_model()`

> Get the original model object.

`Constraint.get_nd()`

> Get the number of dimensions of the constraint.

`Constraint.index(int[] idx)`

> Get a single element from a one-dimensional constraint.

`Constraint.index(int idx)`

> Get a single element from a one-dimensional constraint.

`Constraint.level(int index)`

> Return a single value of the primal solution.

`Constraint.level(int firstidx,int lastidx)`

> Return a slice of the primal solution.

`Constraint.level()`

> Get the primal solution value of the variable.

`Constraint.level(int[] firstidx,int[] lastidx)`

> Return a slice of the primal solution.

`Constraint.toString()`

5.34.1   `SliceConstraint.slice()`

Member of `SliceConstraint`.

```
SliceConstraint.slice(int firstidx,int lastidx)
SliceConstraint.slice(int[] firstidx,int[] lastidx)
```

5.34.1.1   **SliceConstraint.slice(firstidx,lastidx)**

Get a slice of the constraint.

Assuming that the shape of the constraint is one-dimensional, this function creates a constraint slice of (last-first+1) elements, which acts as an interface to a sub-vector of this constraint.

`firstidx : int`

> Index of the first element in the slice.

`lastidx : int`

>   Index if the last element in the slice.

returns : `Constraint`

>   A new constraint object representing a slice of this object.

5.34.1.2   SliceConstraint.slice(firstidx,lastidx)

Get a multi-dimensional slice of the constraint.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the constraint.

`firstidx : int[]`

>   Array of start elements in the slice.

`lastidx : int[]`

>   Array of end element in the slice.

returns : `Constraint`

>   A new constraint object representing a slice of this object.

5.34.2   SliceConstraint.size()

Member of `SliceConstraint`.

`SliceConstraint.size``()`

5.34.2.1   SliceConstraint.size()

Get the total number of elements in the constraint.

returns : `long`

>   The total numbe of elements in the constraint.

5.35   fusion.SliceVariable

An alias for a subset of variables from a single `ModelVariable`.

This class acts as a proxy for accessing a portion of a `ModelVariable`. It is possible to access and modify the properties of the original variable using this alias, and the object can be used in expressions as any other `Variable` object.

Base class

> `Variable`

Known direct descendants

> `BoundInterfaceVariable`

Methods

> `SliceVariable.slice`

Inherited methods

> `Variable.antidiag(int index)`
>> Return an anti-diagonal of a square matrix.
>
> `Variable.antidiag()`
>> Return the anti-diagonal of a square matrix
>
> `Variable.asExpr()`
>> Create an expression corresponding to the variable object.
>
> `Variable.compress()`
>> Reshape a variable object by removing all dimensions of size 1.
>
> `Variable.diag()`
>> Return the diagonal of a square matrix
>
> `Variable.diag(int index)`
>> Return a diagonal of a square matrix.
>
> `Variable.dual(int[] firstidx,int[] lastidx)`
>> Return a slice of the dual solution.
>
> `Variable.dual(int firstidx,int lastidx)`
>> Return a slice of the dual solution.
>
> `Variable.dual()`
>> Get the dual solution value of the variable.
>
> `Variable.index(int[] idx)`
>> Get a single element from a one-dimensional variable.
>
> `Variable.index(int i0,int i1)`
>> Get a single element from a one-dimensional variable.
>
> `Variable.index(int i0,int i1,int i2)`
>> Get a single element from a one-dimensional variable.
>
> `Variable.index(int i0)`
>> Get a single element from a one-dimensional variable.
>
> `Variable.index_flat(long i)`
>> Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`

Return a single value of the primal solution.

`Variable.level()`

Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`

Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`

Return a slice of the primal solution.

`Variable.pick(int[] idxs)`

Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`

Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`

Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

`Variable.size()`

Get the number of elements in the variable.

`Variable.toString()`

Create a string-representation of the variable.

`Variable.transpose()`

Transpose a vector or matrix variable

5.35.1 `SliceVariable.slice()`

Member of `SliceVariable`.

```
SliceVariable.slice(int firstidx,int lastidx)
SliceVariable.slice(int[] firstidx,int[] lastidx)
```

5.35.1.1 **SliceVariable.slice(firstidx,lastidx)**

Get a slice of the variable.

Assuming that the shape of the variable is one-dimensional, this function creates a variable slice of (last-first+1) elements, which acts as an interface to a sub-vector of this variable.

`firstidx : int`

Index of the first element in the slice.

`lastidx : int`

Index if the last element plus one in the slice.

`returns : Variable`

A new variable object representing a slice of this object.

5.35.1.2   SliceVariable.slice(firstidx,lastidx)

Get a multi-dimensional slice of the variable.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the variable.

`firstidx :  int[]`

    Array of start elements in the slice.

`lastidx :   int[]`

    Array of end elements plus one in the slice.

returns : `Variable`

    A new variable object representing a slice of this object.

5.36   `fusion.SparseMatrix`

A two-dimensional specialization if the sparse array.

Base class

    `Matrix`

Constructors

    `SparseMatrix(int dimi,int dimj,int[] subi,int[] subj,double[] val,long nelm)`

Methods

    `SparseMatrix.get`
    `SparseMatrix.getDataAsArray`
        Return the data a dense array of values.
    `SparseMatrix.getDataAsTriplets`
        Return the matrix data in triplet format.
    `SparseMatrix.isSparse`
        Returns true if the matrix is sparse.
    `SparseMatrix.numNonzeros`
        Returns the number of non-zeros in the matrix.
    `SparseMatrix.toString`
        Get a string representation of the matrix.
    `SparseMatrix.transpose`
        Transpose the matrix.

Inherited methods

> `Matrix.numColumns()`
>> Returns the number columns in the matrix.
>
> `Matrix.numRows()`
>> Returns the number rows in the matrix.

5.36.1 `SparseMatrix.SparseMatrix()`

Member of `SparseMatrix`.

`SparseMatrix.SparseMatrix(int dimi,int dimj,int[] subi,int[] subj,double[] val,long nelm)`

5.36.1.1 SparseMatrix.SparseMatrix(dimi,dimj,subi,subj,val,nelm)

Construct a sparse matrix from sparse data. The input data need not be sorted and may contain duplicate entries (which will be added up).

The input subscripts are verified.

`dimi : int`
> Number of rows in the matrix, must be strictly positive.

`dimj : int`
> Number of columns in the matrix, must be strictly positive.

`subi : int[]`
> Row subscripts of non-zero elements.

`subj : int[]`
> Column subscripts of non-zero elements.

`val : double[]`
> Values of non-zero elements.

`nelm : long`
> Number of elements. The subi, subj and val arrays must be at least this long.

5.36.2 `SparseMatrix.get()`

Member of `SparseMatrix`.

`SparseMatrix.get(int i,int j)`

5.36.2.1   SparseMatrix.get(i,j)

```
i :   int
```

```
j :   int
```

```
returns : double
```

5.36.3   SparseMatrix.transpose()

Member of SparseMatrix.
```
SparseMatrix.transpose()
```

5.36.3.1   SparseMatrix.transpose()

Transpose the matrix.

```
returns : Matrix
```

5.36.4   SparseMatrix.numNonzeros()

Member of SparseMatrix.
```
SparseMatrix.numNonzeros()
```

5.36.4.1   SparseMatrix.numNonzeros()

Returns the number of non-zeros in the matrix.

```
returns : long
```
       The number of non-zeros.

5.36.5   SparseMatrix.isSparse()

Member of SparseMatrix.
```
SparseMatrix.isSparse()
```

5.36.5.1   SparseMatrix.isSparse()

Returns true if the matrix is sparse.

```
returns : boolean
```

5.36.6 SparseMatrix.toString()

Member of SparseMatrix.

`SparseMatrix.toString()`

5.36.6.1 SparseMatrix.toString()

Get a string representation of the matrix.

returns : `String`

A string representation of the matrix.

5.36.7 SparseMatrix.getDataAsTriplets()

Member of SparseMatrix.

`SparseMatrix.getDataAsTriplets(int[] subi_,int[] subj_,double[] cof_)`

5.36.7.1 SparseMatrix.getDataAsTriplets(subi_,subj_,cof_)

Return the matrix data in triplet format. Data is copied to the arrays subi, subj and val which must be allocated to hold at least the number of non-zeros in the matrix.

The data returned must be ordered with `subi` as primary key and `subj` as secondary key.

subi_ : `int[]`

Row subscripts are returned in this array.

subj_ : `int[]`

Column subscripts are returned in this array.

cof_ : `double[]`

Coefficient values are returned in this arary.

5.36.8 SparseMatrix.getDataAsArray()

Member of SparseMatrix.

`SparseMatrix.getDataAsArray()`

5.36.8.1 SparseMatrix.getDataAsArray()

Return the matrix elements as an array. If it is a sparse matrix, it is converted returned as a dense matrix, so the size of the returned array is always exactly the size of the matrix.

returns : `double[]`

5.37   `fusion.StringSet`

A class defining a one-dimensional set of strings.

Base class

> `BaseSet`

Constructors

> `StringSet(String[] ks)`
> > Create a set of unique strings.

Methods

> `StringSet.getname`
>
> `StringSet.indexToString`
>
> `StringSet.slice`
>
> `StringSet.stride`
> > Return the stride size in the given dimension.
>
> `StringSet.toString`
> > Return a string representation of the set.

Inherited methods

> `Set.compare(Set other)`
> > Compare two sets and return true if they have the same shape and size.
>
> `BaseSet.dim(int i)`
> > Return the size of the given dimension.
>
> `Set.getSize()`
> > Total number of elements in the set.
>
> `Set.idxtokey(long idx)`
> > Convert a linear index to a N-dimensional key.
>
> `Set.realnd()`
> > Number of dimensions of more than 1 element, or 1 if the number of significant dimensions is 0.

5.37.1   `StringSet.StringSet()`

Member of `StringSet`.

`StringSet.StringSet(String[] ks)`

5.37.1.1 StringSet.StringSet(ks)

Create a set of unique strings.

```
ks :  String[]
```
   List of unique strings

5.37.2 StringSet.stride()

Member of StringSet.
```
StringSet.stride(int i)
```


5.37.2.1 StringSet.stride(i)

Return the stride size in the given dimension.

```
i :  int
```
   Dimension index.

```
returns : long
```
   The stride size in the requested dimension.

5.37.3 StringSet.getname()

Member of StringSet.
```
StringSet.getname(long key)
StringSet.getname(int[] key)
```


5.37.3.1 StringSet.getname(key)

Return a string representing the index.

```
key :  long
```
   A linear index.

```
returns : String
```
   Get a string representing the item identified by the key.

5.37.3.2 StringSet.getname(key)

Return a string representing the index.

`key :  int[]`

>   An N-dimensional index.

`returns : String`

>   Get a string representing the item identified by the key.

5.37.4 StringSet.slice()

Member of StringSet.

`StringSet.slice(int first_,int last_)`
`StringSet.slice(int[] first_,int[] last_)`

5.37.4.1   StringSet.slice(first_,last_)

Create a set object representing a slice of this set.

`first_ :  int`

>   Last index in the range.

`last_ :  int`

`returns : Set`

>   A new Set object representing the slice.

5.37.4.2   StringSet.slice(first_,last_)

Create a set object representing a slice of this set.

`first_ :  int[]`

>   Last index in each dimension in the range.

`last_ :  int[]`

`returns : Set`

>   A new Set object representing the slice.

5.37.5 StringSet.toString()

Member of StringSet.

`StringSet.toString()`

5.37.5.1   StringSet.toString()

Return a string representation of the set.

 returns : `String`

A string representation of the set.

5.37.6   `StringSet.indexToString()`

Member of `StringSet`.

 `StringSet.indexToString(long index)`

5.37.6.1   StringSet.indexToString(index)

 index :   `long`

 returns : `String`

5.38   `fusion.SymmetricVariable`

In some cases using this variable instead if a normal Variable can reduce the number of instantiated variables by approximately 50%.

It can be constructed in various ways: Either from one existing NxN variable, creating a new variable that mirrors the lower triangular part of that variable, or from multiple variables.

Use the Variable.symmetric() functions to construct symmetric variables.

 Base class

`Variable`

 Methods

`SymmetricVariable.slice`

 Inherited methods

`Variable.antidiag(int index)`
    Return an anti-diagonal of a square matrix.
`Variable.antidiag()`
    Return the anti-diagonal of a square matrix
`Variable.asExpr()`
    Create an expression corresponding to the variable object.

`Variable.compress()`

    Reshape a variable object by removing all dimensions of size 1.

`Variable.diag()`

    Return the diagonal of a square matrix

`Variable.diag(int index)`

    Return a diagonal of a square matrix.

`Variable.dual(int[] firstidx,int[] lastidx)`

    Return a slice of the dual solution.

`Variable.dual(int firstidx,int lastidx)`

    Return a slice of the dual solution.

`Variable.dual()`

    Get the dual solution value of the variable.

`Variable.index(int[] idx)`

    Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1)`

    Get a single element from a one-dimensional variable.

`Variable.index(int i0,int i1,int i2)`

    Get a single element from a one-dimensional variable.

`Variable.index(int i0)`

    Get a single element from a one-dimensional variable.

`Variable.index_flat(long i)`

    Index into the variable as if it was a one-dimensional object.

`Variable.level(int index)`

    Return a single value of the primal solution.

`Variable.level()`

    Get the primal solution value of the variable.

`Variable.level(int[] firstidx,int[] lastidx)`

    Return a slice of the primal solution.

`Variable.level(int firstidx,int lastidx)`

    Return a slice of the primal solution.

`Variable.pick(int[] idxs)`

    Create a vector-variable by picking a list of indexes from this variable.

`Variable.pick(int[][] midxs)`

    Create a matrix-variable by picking a list of indexes from this variable.

`Variable.pick_flat(long[] indexes)`

    Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

```
Variable.size()
```
Get the number of elements in the variable.

```
Variable.toString()
```
Create a string-representation of the variable.

```
Variable.transpose()
```
Transpose a vector or matrix variable

5.38.1   `SymmetricVariable.slice()`

Member of `SymmetricVariable`.

```
SymmetricVariable.slice(int first,int last)
SymmetricVariable.slice(int[] first,int[] last)
```

5.38.1.1   SymmetricVariable.slice(first,last)

Get a slice of the variable.

Assuming that the shape of the variable is one-dimensional, this function creates a variable slice of (last-first+1) elements, which acts as an interface to a sub-vector of this variable.

```
first :   int
```
Index of the first element in the slice.

```
last :   int
```
Index if the last element plus one in the slice.

returns : `Variable`

A new variable object representing a slice of this object.

5.38.1.2   SymmetricVariable.slice(first,last)

Get a multi-dimensional slice of the variable.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the variable.

```
first :   int[]
```
Array of start elements in the slice.

```
last :   int[]
```
Array of end elements plus one in the slice.

returns : `Variable`

A new variable object representing a slice of this object.

5.39 `fusion.Variable`

An abstract variable object. This is the base class for all variable types in Fusion, and it contains several static methods for manipulating variable objects.

The `Variable` object can be an interface to the normal model variables, e.g. `LinearVariable` and `ConicVariable`, to slices of other variables or to concatenations of other variables.

Primal and dual solution values can be accessed through the `Variable` object.

Known direct descendants

>   `ModelVariable`, `CompoundVariable`, `SliceVariable`, `SymmetricVariable`

Constructors

>   `Variable(Model m,Set shape)`
>   Create a variable object.

Methods

>   `Variable.antidiag`
>   Return the antidiagonal of a square variable matrix.

>   `Variable.asExpr`
>   Create an expression corresponding to the variable object.

>   `Variable.compress`
>   Reshape a variable object by removing all dimensions of size 1.

>   `Variable.diag`
>   Return the diagonal of a square variable matrix.

>   `Variable.dual`
>   Return the dual value of the variable as an array.

>   `Variable.index`
>   Return a variable slice of size 1 corresponding to a single element in the variable object..

>   `Variable.index_flat`
>   Index into the variable as if it was a one-dimensional object.

>   `Variable.level`
>   Return the primal value of the variable as an array.

>   `Variable.pick`
>   Create a slice variable by picking a list of indexes from this variable.

>   `Variable.pick_flat`
>   Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

>   `Variable.size`
>   Get the number of elements in the variable.

Variable.slice

Variable.toString
>   Create a string-representation of the variable.

Variable.transpose
>   Transpose a vector or matrix variable

Static Methods

Variable.flatten

Variable.hstack
>   Create a stacked variable in second dimension.

Variable.repeat
>   Create a variable by stacking v on top of itself n times.

Variable.reshape

Variable.stack
>   Create a stacked variable.

Variable.symmetric

Variable.vstack
>   Create a stacked variable in first dimension.

5.39.1   Variable.Variable()

Member of Variable.

```
Variable.Variable(Model m,Set shape)
```

5.39.1.1   Variable.Variable(m,shape)

Create a variable object.

m :   Model
>   The model to which this variable belongs.

shape :   Set
>   The shape of the variable.

5.39.2   Variable.size()

Member of Variable.

```
Variable.size()
```

5.39.2.1   Variable.size()

Get the number of elements in the variable.

 returns : `long`

5.39.3   `Variable.index()`

Member of `Variable`.

```
Variable.index(int i0)
Variable.index(int i0,int i1)
Variable.index(int i0,int i1,int i2)
Variable.index(int[] idx)
```

Return a variable slice of size 1 corresponding to a single element in the variable object..

 i0 : int

    Index in the first dimension of the element requested.

 i1 : int

    Index in the second dimension of the element requested.

 idx : int[]

    Index of the element requested.

5.39.3.1   Variable.index(i0)

Get a single element from a one-dimensional variable.

 i0 :   int

    Index in the first dimension of the element requested.

 returns : `Variable`

    A new slice containing a single element.

5.39.3.2   Variable.index(i0,i1)

Get a single element from a one-dimensional variable.

 i0 :   int

    Index in the first dimension of the element requested.

 i1 :   int

    Index in the second dimension of the element requested.

 returns : `Variable`

    A new slice containing a single element.

5.39.3.3   Variable.index(i0,i1,i2)

Get a single element from a one-dimensional variable.

`i0 :  int`

   Index in the first dimension of the element requested.

`i1 :  int`

   Index in the second dimension of the element requested.

`i2 :  int`

returns : `Variable`

   A new slice containing a single element.

5.39.3.4   Variable.index(idx)

Get a single element from a one-dimensional variable.

`idx :  int[]`

   Array of integers entry in each dimension.

returns : `Variable`

   A new slice containing a single element.

5.39.4   `Variable.slice()`

Member of `Variable`.

```
abstract Variable.slice(int first,int last)
abstract Variable.slice(int[] first,int[] last)
```

5.39.4.1   abstract Variable.slice(first,last)

Get a slice of the variable.

Assuming that the shape of the variable is one-dimensional, this function creates a variable slice of (last-first+1) elements, which acts as an interface to a sub-vector of this variable.

`first :  int`

   Index of the first element in the slice.

`last :  int`

   Index if the last element plus one in the slice.

returns : `Variable`

   A new variable object representing a slice of this object.

5.39.4.2   abstract Variable.slice(first,last)

Get a multi-dimensional slice of the variable.

The two arrays define the first and last element (both inclusive) in each dimension of the slice. The arrays must have the same length as the variable.

`first :  int[]`

> Array of start elements in the slice.

`last :  int[]`

> Array of end elements plus one in the slice.

returns : `Variable`

> A new variable object representing a slice of this object.

5.39.5   `Variable.dual()`

Member of `Variable`.

```
Variable.dual(int firstidx,int lastidx)
Variable.dual(int[] firstidx,int[] lastidx)
Variable.dual()
```

Return the dual value of the variable as an array.

firstidx : int[]

> Index of the first value requested.

lastidx : int[]

> Index of the last-plus-one value requested.

5.39.5.1   Variable.dual(firstidx,lastidx)

Return a slice of the dual solution from a one-dimensional variable. If the variable is not one-dimensional, an exception will be thrown. The solution values are taken from the default solution defined in the `Model`.

`firstidx :   int`

> Index of the first element in the range.

`lastidx :   int`

> Index of the last element (inclusive) in the range.

Throws exceptions :

- `SolutionError`

returns : `double[]`

>   An array of solution values.

5.39.5.2 Variable.dual(firstidx,lastidx)

Return a slice of the dual solution from a variable. If the length of the index arrays does not match the number of dimensions in the variable, an exception will be thrown. The solution values are taken from the default solution defined in the `Model`.

`firstidx :   int[]`

>   Array of indexes of the first element in each dimension.

`lastidx :   int[]`

>   Array of indexes of the last element (inclusive) in each dimension.

Throws exceptions :

>   - `SolutionError`

returns : `double[]`

>   An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

5.39.5.3 Variable.dual()

Get the dual solution value of the variable.

returns : `double[]`

>   An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

5.39.6 `Variable.hstack()`

Member of `Variable`.
```
static Variable.hstack(Variable[] v)
static Variable.hstack(Variable v0,Variable v1)
static Variable.hstack(Variable v0,Variable v1,Variable v2)
```

Create a stacked variable in second dimension.

`v :` `Variable[]`

>   List of variable to stack.

`v0 :` `Variable`

>   First variable in the stack.

v1 : Variable

    second variable in the stack.

v2 : Variable

    Third variable in the stack.

5.39.6.1   static Variable.hstack(v)

Create a stacked variable.

Create a new variable representing the concatenation of the given variables in second dimension. This works the same way as for two two variables. All variables must have the same number of dimensions, and all dimensions except the second must have the same size. Entries that are `null` are ignored.

v :   `Variable[]`

    List of variable to stack.

returns : `Variable`

    An object representing the concatenation of the variables.

5.39.6.2   static Variable.hstack(v0,v1)

Stack two variables.

Create a new variable representing the concatenation of the given variables. The two variables must have the same number of dimensions, and the size of each dimension must match, except for the second.

The stack of two variables v_a and v_b with dimensions $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$ is a variable with of dimension k where $d_2 = a_2 + b_2$ and $d_i = a_i = b_i$ for $i = 1, 3, 4, \ldots, k$.

v0 :   `Variable`

    First variable in the stack.

v1 :   `Variable`

    The first variable in the stack.

returns : `Variable`

    An object representing the concatenation of the variables.

5.39.6.3   static Variable.hstack(v0,v1,v2)

Stack three variables.

Create a new variable representing the concatenation of the given variables in second dimension. This works the same way as for two two variables. The operands must have the same number of dimensions and all dimensions except the second must have the same size.

v0 : `Variable`

> First variable in the stack.

v1 : `Variable`

> The first variable in the stack.

v2 : `Variable`

> The second variable in the stack.

returns : `Variable`

> An object representing the concatenation of the variables.

5.39.7 `Variable.level()`

Member of `Variable`.

```
Variable.level(int firstidx,int lastidx)
Variable.level(int[] firstidx,int[] lastidx)
Variable.level(int index)
Variable.level()
```

Return the primal value of the variable as an array.

firstidx : int[]

> Index of the first value requested.

lastidx : int[]

> Index of the last-plus-one value requested.

5.39.7.1 **Variable.level(firstidx,lastidx)**

Return a slice of the primal solution from a one-dimensional variable. If the variable is not one-dimensional, an exception will be thrown. The solution values are taken from the default solution defined in the `Model`.

firstidx : int

> Index of the first element in the range.

lastidx : int

> Index of the last element (inclusive) in the range.

Throws exceptions :

> - `SolutionError`

returns : `double[]`

> An array of solution values.

5.39.7.2   Variable.level(firstidx,lastidx)

Return a slice of the primal solution from a variable. If the length of the index arrays does not match the number of dimensions in the variable, an exception will be thrown. The solution values are taken from the default solution defined in the `Model`.

`firstidx :  int[]`

> Array of indexes of the first element in each dimension.

`lastidx :  int[]`

> Array of indexes of the last element (inclusive) in each dimension.

> Throws exceptions :

> > • `SolutionError`

`returns : double[]`

> An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

5.39.7.3   Variable.level(index)

Return a single solution value from a 1-dimensional variable. This corresponds to getting a 1-dimensionals slice of size 1.

See `Variable.level` for details.

`index :  int`

> Index of the element whose solution value to return.

> Throws exceptions :

> > • `SolutionError`

`returns : double`

> The solution value as a double.

5.39.7.4   Variable.level()

Get the primal solution value of the variable.

> Throws exceptions :

> > • `SolutionError`

`returns : double[]`

> An array of solution values. When the selected slice is multi-dimensional, this corresponds to the flattened slice of solution values.

5.39.8 `Variable.index_flat()`

Member of `Variable`.

`Variable.index_flat(long i)`

5.39.8.1 Variable.index_flat(i)

Index into the variable as if it was a one-dimensional object.

`i : long`

`returns : Variable`

5.39.9 `Variable.diag()`

Member of `Variable`.

`Variable.diag(int index)`
`Variable.diag()`

Return the diagonal of a square variable matrix.

index : int

Index of the anti-diagonal

5.39.9.1 Variable.diag(index)

Return a diagonal of a square matrix.

`index : int`

Defining the index of the diagonal. 0 is the diagonal starting at element (1,1). Positive values are the super-diagonals (diagonals in the upper triangular part, and negative are indexes of the sub-diagonals (in the lower triangular part).

`returns : Variable`

5.39.9.2 Variable.diag()

Return the diagonal of a square matrix

`returns : Variable`

5.39.10 `Variable.compress()`

Member of `Variable`.

`Variable.compress()`

5.39.10.1   Variable.compress()

Reshape a variable object by removing all dimensions of size 1. The result contains the same number of elements, but all dimensions are larger than 1 (except if the original variable contains exactly one element).

returns : `Variable`

5.39.11   Variable.transpose()

Member of `Variable`.

`Variable.transpose()`

5.39.11.1   Variable.transpose()

Note that this requires a one- or two-dimensional variable.

returns : `Variable`

5.39.12   Variable.pick_flat()

Member of `Variable`.

`Variable.pick_flat(long[] indexes)`

5.39.12.1   Variable.pick_flat(indexes)

Create a vector-variables by picking a list of indexes from this variable as if it was a one-dimensional object.

indexes : `long[]`

returns : `Variable`

5.39.13   Variable.symmetric()

Member of `Variable`.

`static Variable.symmetric(Variable v)`

5.39.13.1   static Variable.symmetric(v)

v : `Variable`

returns : `Variable`

5.39.14 `Variable.repeat()`

Member of `Variable`.

` static Variable.repeat(Variable v,int num)`

5.39.14.1   static Variable.repeat(v,num)

If `v` is an (d)-dimensional object, then the returned object will be a (d+1) dimensional object, and the size of the last dimension is (num).

` v :   Variable`

>   A Variable object.

` num :   int`

>   The number of times to repeat the variable.

` returns : Variable`

5.39.15 `Variable.toString()`

Member of `Variable`.

` Variable.toString()`

5.39.15.1   Variable.toString()

Create a string-representation of the variable.

` returns : String`

>   A string representing the variable.

5.39.16 `Variable.antidiag()`

Member of `Variable`.

` Variable.antidiag(int index)`
` Variable.antidiag()`

Return the antidiagonal of a square variable matrix.

` index : int`

>   Index of the anti-diagonal

5.39.16.1   Variable.antidiag(index)

Return an anti-diagonal of a square matrix.

```
index :   int
```
>   Defining the index of the anti-diagonal. 0 is the diagonal starting at element (1,n). Positive values are the super-diagonals (diagonals in the upper triangular part, and negative are indexes of the sub-diagonals (in the lower triangular part).

   returns : `Variable`

5.39.16.2   Variable.antidiag()

Return the anti-diagonal of a square matrix

   returns : `Variable`

5.39.17   Variable.reshape()

Member of `Variable`.

```
static Variable.reshape(Variable v,Set s)
static Variable.reshape(Variable v,int d1,int d2)
static Variable.reshape(Variable v,int d1)
```

5.39.17.1   static Variable.reshape(v,s)

Create a copy of the variable `v` with a new shape `s`. The total size of the new shape must be identical to the size of the variable.

For example, reshaping a two-dimensional variable V with the dimensions (d_1, d_2) into a one-dimensional variable of size $d_1 \cdot d_2$ will result in a variable vector

$$[V_{1,1}, \ldots, V_{1,d_2}, V_{2,1} \ldots, V_{2,d_2}, \ldots, V_{d_1,1}, \ldots, V_{d_1,d_2}]$$

Similarly, if a variable W is one-dimensional with size $d_1 \cdot d_2$, then it can be reshaped to a two-dimensional variable if size (d_1, d_2). This would result in a variable of the form

$$\begin{bmatrix} W_1 & \cdots & W_{d_2} \\ W_{d_2+1} & \cdots & W_{2d_2} \\ \\ W_{(d_1-1)*d_2+1} & \cdots & W_{d_1 \cdot d_2} \end{bmatrix}$$

```
v :   Variable
```
>   The variable to be reshaped

s : `Set`

> The new shape of the variable

returns : `Variable`

> A new variable object with the shape defined by `s`

5.39.17.2   static Variable.reshape(v,d1,d2)

Reshape a Variable object into a two-dimensional variable object.

Note that the total number of elements in the result must be the same as in the variable.

v : `Variable`

> The variable object to reshape.

d1 : `int`

> Size of first dimension in the result.

d2 : `int`

> Size of second dimension in result.

returns : `Variable`

5.39.17.3   static Variable.reshape(v,d1)

Reshape a Variable object into a one-dimensional variable object.

Note that the total number of elements in the result must be the same as in the variable.

v : `Variable`

> The variable object to reshape.

d1 : `int`

> Size of first dimension in the result.

returns : `Variable`

5.39.18   `Variable.pick()`

Member of `Variable`.

`Variable.pick(int[] idxs)`
`Variable.pick(int[][] midxs)`

Create a slice variable by picking a list of indexes from this variable.

idxs : int[]

    Indexes of the elements requested.

midxs : int[][]

    Matrix of indexes of the elements requested.

5.39.18.1   Variable.pick(idxs)

Create a vector-variable by picking a list of indexes from this variable.

`idxs  :  int[]`

    Indexes of the elements requested.

`returns : Variable`

5.39.18.2   Variable.pick(midxs)

Create a matrix-variable by picking a list of indexes from this variable.

`midxs  :  int[][]`

    Matrix of indexes of the elements requested.

`returns : Variable`

5.39.19   Variable.asExpr()

Member of `Variable`.

`Variable.asExpr()`

5.39.19.1   Variable.asExpr()

Create an `Expression` object corresponding to $I \cdot V$, where I is the identity matrix and V is this variable.

`returns : Expression`

    An Expression object representing the product I*V, where I is the identity matrix and V is this variable.

5.39.20   Variable.vstack()

Member of `Variable`.

`static Variable.vstack(Variable[] v)`

```
static Variable.vstack(Variable v0,Variable v1)
static Variable.vstack(Variable v0,Variable v1,Variable v2)
```

Create a stacked variable in first dimension.

v : Variable[]

> List of variable to stack.

v0 : Variable

> First variable in the stack.

v1 : Variable

> second variable in the stack.

v2 : Variable

> Third variable in the stack.

5.39.20.1   static Variable.vstack(v)

Create a stacked variable.

Create a new variable representing the concatenation of the given variables. This works the same way as for two two variables. All variables must have the same number of dimensions, and all dimensions except the first must have the same size. Entries that are `null` are ignored.

v :   `Variable[]`

> List of variable to stack.

returns : `Variable`

> An object representing the concatenation of the variables.

5.39.20.2   static Variable.vstack(v0,v1)

Stack two variables.

Create a new variable representing the concatenation of the given variables. The two variables must have the same number of dimensions, and the size of each dimension must match.

The stack of two variables v_a and v_b with dimensions $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$ is a variable with of dimension k where $d_1 = a_1 + b_1$ and $d_i = a_i = b_i$ for $i > 1$.

v0 :   `Variable`

> First variable in the stack.

v1 :   `Variable`

> The first variable in the stack.

returns : `Variable`

> An object representing the concatenation of the variables.

5.39.20.3   static Variable.vstack(v0,v1,v2)

Stack three variables.

Create a new variable representing the concatenation of the given variables. This works the same way as for two two variables. The operands must have the same number of dimensions, and all dimensions except the first must have the same size.

 v0 :   `Variable`

    First variable in the stack.

 v1 :   `Variable`

    The first variable in the stack.

 v2 :   `Variable`

    The second variable in the stack.

 returns : `Variable`

    An object representing the concatenation of the variables.

5.39.21   Variable.stack()

Member of `Variable`.

`static Variable.stack(Variable[][] vlist)`

5.39.21.1   static Variable.stack(vlist)

Create a stacked variable.

This creates a two-dimensional stacking of variables. A list of variables

$$\{v_{ij} :\ i = 1 \ldots m,\ j = 1 \ldots n_i\}$$

must satisfy the following conditions to define a valid stacking:

- The number of dimensions for all v_{ij} must be the same, with one exception: A variables of size n is treated as if it had size $n{\times}1$

- For each row, the first dimension of all variables must be the same., i.e. $d_{ij}^1 = H_i$ for some constant R_i for all i, j.

- The sum of the second dimension of the variables must be the same for all rows, i.e.

$$\sum_{j=1}^{n_i} d_{ij}^2 = W \text{ for } i = 1 \ldots m$$

    for some constant W.

- All remaining dimension (three and up) must be the same, i.e.

$$d_{ij}^k = D_k$$

for constants D_k.

`vlist :` `Variable[][]`
The variables in the stack.

`returns :` `Variable`
An object representing the concatenation of the variables.

5.39.22 `Variable.flatten()`

Member of `Variable`.

`static Variable.flatten(Variable v)`

5.39.22.1   static Variable.flatten(v)

`v :` `Variable`

`returns :` `Variable`

5.40   Enums

5.40.1   Enum `fusion.AccSolutionStatus`

Constants used for defining which solutions statuses are acceptable.

Enum members:

Anything
Accept all solution status except `SolutionStatus.Undefined`.

Optimal
Accept only optimal solution status.

NearOptimal

Feasible
Accept any feasible solution, even if not optimal.

Certificate
Accept only a certificate.

5.40.2   Enum `fusion.ObjectiveSense`

Used in `Model.objective` to define the objective sense of the `Model`.

Enum members:

Undefined

    The sense is not defined; trying to optimize a `Model` whose objective sense is undefined is an error.

Minimize

    Minimize the objective.

Maximize

    Maximize the objective.

5.40.3   Enum `fusion.PSDKey`

Enum members:

IsSymPSD

IsTrilPSD

IsLinPSD

5.40.4   Enum `fusion.RelationKey`

Used internally in Fusion to define the domain type for a constraint or variable.

Enum members:

EqualsTo

LessThan

GreaterThan

IsFree

InQCone

InRotatedQCone

InRange

5.40.5 Enum `fusion.SolutionStatus`

Defines properties of either a primal or a dual solution. A model may contain multiple solutions which may have different status.

Specifically, there will be individual solutions, and thus solution statuses, for the interior-point, simplex and integer solvers.

Enum members:

Undefined

Undefined solution. This means that no values exist for the relevant solution.

Unknown

The solution status is unknown; this will happen if the user inputs values or a solution is read from a file.

Optimal

The solution values are feasible and optimal.

NearOptimal

The solution values are feasible and nearly optimal.

Feasible

The solution is feasible.

NearFeasible

The solution is nearly feasible.

Certificate

The solution is a certificate of infeasibility (primal or dual, depending no which solution it belongs to).

NearCertificate

The solution is nearly a certificate of infeasibility (primal or dual, depending no which solution it belongs to).

5.40.6 Enum `fusion.SolutionType`

Used when requesting a specific solution from a `Model`.

Enum members:

Default

Auto-select the default solution; usually this will be the integer solution, if available, otherwise the basic solution, if available, otherwise the interior-point solution.

Basic

    Select the basic solution.

Interior

    Select the interior-point solution.

Integer

    Select the integer solution.

5.40.7   Enum `fusion.StatusKey`

Defines the status of a single solution value.

Enum members:

Unknown

    The status is unknown; this will happen if, for example, the solution was read from a file or inputted by the user.

Basic

    The solution is basic.

SuperBasic

    The value is superbasic.

OnBound

    The value is on its bound.

Infinity

    The solution value is infinite, or sufficiently large to be deemed infinite.

5.41   Exceptions

5.41.1   `fusion.DimensionError`

Thrown when an operation required an object with a specific number of dimensions, or dimensions of a specific size, but got something else.

Base class

    FusionRuntimeException

Constructors

    DimensionError(String msg)

Inherited methods

> `FusionRuntimeException.toString()`

5.41.1.1 `DimensionError.DimensionError()`

Member of `DimensionError`.

`DimensionError.DimensionError(String msg)`

DimensionError.DimensionError(msg)

`msg : String`

5.41.2 `fusion.DomainError`

Invalid domain.

Base class

> `FusionRuntimeException`

Constructors

> `DomainError(String msg)`

Inherited methods

> `FusionRuntimeException.toString()`

5.41.2.1 `DomainError.DomainError()`

Member of `DomainError`.

`DomainError.DomainError(String msg)`

DomainError.DomainError(msg)

`msg : String`

5.41.3 `fusion.ExpressionError`

Tried to construct an expression from invalid.

Base class

> `FusionRuntimeException`

Constructors

> ExpressionError(String msg)

Inherited methods

> FusionRuntimeException.toString()

5.41.3.1 ExpressionError.ExpressionError()

Member of ExpressionError.

ExpressionError.ExpressionError(String msg)

ExpressionError.ExpressionError(msg)

msg : String

5.41.4 fusion.FatalError

Constructors

> FatalError(String msg)

5.41.4.1 FatalError.FatalError()

Member of FatalError.

FatalError.FatalError(String msg)

FatalError.FatalError(msg)

msg : String

5.41.5 fusion.FusionException

Base class for all normal exceptions in Fusion.

Known direct descendants

> SolutionError

Constructors

> FusionException(String msg_)

Methods

> FusionException.toString

5.41.5.1   FusionException.FusionException()

Member of FusionException.

FusionException.FusionException(String msg_)

FusionException.FusionException(msg_)

 msg_ :   String

5.41.5.2   FusionException.toString()

Member of FusionException.

FusionException.toString()

FusionException.toString()

 returns : String

5.41.6   fusion.FusionRuntimeException

Base class for all run-time exceptions in Fusion.

 Known direct descendants

  MatrixError, DomainError, OptimizeError, SetDefinitionError, DimensionError, ValueConversionError, ParameterError, RangeError, LengthError, ModelError, SparseFormatError, IOError, NameError, SliceError, ExpressionError, IndexError

 Constructors

  FusionRuntimeException(String msg_)

 Methods

  FusionRuntimeException.toString

5.41.6.1   FusionRuntimeException.FusionRuntimeException()

Member of FusionRuntimeException.

FusionRuntimeException.FusionRuntimeException(String msg_)

FusionRuntimeException.FusionRuntimeException(msg_)

 msg_ :   String

5.41.6.2 `FusionRuntimeException.toString()`

Member of `FusionRuntimeException`.
`FusionRuntimeException.toString()`

FusionRuntimeException.toString()

 returns : `String`

5.41.7   `fusion.IOError`

Error when reading or writing a stream, or opening a file.

 Base class

   `FusionRuntimeException`

 Constructors

   `IOError(String msg)`

 Inherited methods

   `FusionRuntimeException.toString()`

5.41.7.1 `IOError.IOError()`

Member of `IOError`.
`IOError.IOError(String msg)`

IOError.IOError(msg)

 msg :   `String`

5.41.8   `fusion.IndexError`

Index out of bound, or a multi-dimensional index had wrong number of dimensions.

 Base class

   `FusionRuntimeException`

 Constructors

   `IndexError(String msg)`

Inherited methods

FusionRuntimeException.toString()

5.41.8.1 IndexError.IndexError()

Member of IndexError.

IndexError.IndexError(String msg)

IndexError.IndexError(msg)

msg :   String

5.41.9 fusion.LengthError

An array did not have the required length, or two arrays were expected to have same length.

Base class

FusionRuntimeException

Constructors

LengthError(String msg)

Inherited methods

FusionRuntimeException.toString()

5.41.9.1 LengthError.LengthError()

Member of LengthError.

LengthError.LengthError(String msg)

LengthError.LengthError(msg)

msg :   String

5.41.10 fusion.MatrixError

Thrown if data used in construction of a matrix contained inconsistencies or errors.

Base class

FusionRuntimeException

Constructors

     `MatrixError(String msg)`

Inherited methods

     `FusionRuntimeException.toString()`

5.41.10.1 `MatrixError.MatrixError()`

Member of `MatrixError`.

`MatrixError.MatrixError(String msg)`

MatrixError.MatrixError(msg)

`msg :  String`

5.41.11   `fusion.ModelError`

Thrown when objects from different models were mixed.

Base class

     `FusionRuntimeException`

Constructors

     `ModelError(String msg)`

Inherited methods

     `FusionRuntimeException.toString()`

5.41.11.1 `ModelError.ModelError()`

Member of `ModelError`.

`ModelError.ModelError(String msg)`

ModelError.ModelError(msg)

`msg :  String`

5.41.12 `fusion.NameError`

Name clash; tries to add a variable or constraint with a name that already exists.

Base class

FusionRuntimeException

Constructors

NameError(String msg)

Inherited methods

FusionRuntimeException.toString()

5.41.12.1 `NameError.NameError()`

Member of NameError.

NameError.NameError(String msg)

NameError.NameError(msg)

msg : String

5.41.13 `fusion.OptimizeError`

An error occurred during optimization.

Base class

FusionRuntimeException

Constructors

OptimizeError(String msg)

Inherited methods

FusionRuntimeException.toString()

5.41.13.1 `OptimizeError.OptimizeError()`

Member of OptimizeError.

OptimizeError.OptimizeError(String msg)

OptimizeError.OptimizeError(msg)

 msg : String

5.41.14   fusion.ParameterError

Tried to use an invalid parameter for a value that was invalid for a specific parameter.

 Base class

    FusionRuntimeException

 Constructors

     ParameterError(String msg)

 Inherited methods

    FusionRuntimeException.toString()

5.41.14.1   ParameterError.ParameterError()

Member of ParameterError.

 ParameterError.ParameterError(String msg)

ParameterError.ParameterError(msg)

 msg : String

5.41.15   fusion.RangeError

Invalid range specified

 Base class

    FusionRuntimeException

 Constructors

     RangeError(String msg)

 Inherited methods

    FusionRuntimeException.toString()

5.41.15.1 `RangeError.RangeError()`

Member of `RangeError`.

`RangeError.RangeError(String msg)`

RangeError.RangeError(msg)

`msg :  String`

5.41.16 `fusion.SetDefinitionError`

Invalid data for constructing set.

Base class

`FusionRuntimeException`

Constructors

`SetDefinitionError(String msg)`

Inherited methods

`FusionRuntimeException.toString()`

5.41.16.1 `SetDefinitionError.SetDefinitionError()`

Member of `SetDefinitionError`.

`SetDefinitionError.SetDefinitionError(String msg)`

SetDefinitionError.SetDefinitionError(msg)

`msg :  String`

5.41.17 `fusion.SliceError`

Invalid slice definition, negative slice or slice index out of bounds.

Base class

`FusionRuntimeException`

Constructors

`SliceError(String msg)`

    SliceError()

 Inherited methods

    FusionRuntimeException.toString()

5.41.17.1 SliceError.SliceError()

Member of SliceError.

    SliceError.SliceError(String msg)
    SliceError.SliceError()

SliceError.SliceError(msg)

 msg : String

SliceError.SliceError()

5.41.18   fusion.SolutionError

Requested a solution that was undefined or whose status was not acceptable.

 Base class

    FusionException

 Constructors

    SolutionError(String msg)

    SolutionError()

 Inherited methods

    FusionException.toString()

5.41.18.1 SolutionError.SolutionError()

Member of SolutionError.

    SolutionError.SolutionError(String msg)
    SolutionError.SolutionError()

SolutionError.SolutionError(msg)

 msg : String

SolutionError.SolutionError()

5.41.19 `fusion.SparseFormatError`

The given sparsity patters was invalid or specified an index that was out of bounds.

Base class

> `FusionRuntimeException`

Constructors

> `SparseFormatError(String msg)`

Inherited methods

> `FusionRuntimeException.toString()`

5.41.19.1 `SparseFormatError.SparseFormatError()`

Member of `SparseFormatError`.

`SparseFormatError.SparseFormatError(String msg)`

SparseFormatError.SparseFormatError(msg)

`msg : String`

5.41.20 `fusion.UnexpectedError`

Constructors

> `UnexpectedError(String msg)`
>
> `UnexpectedError(FusionException e)`

5.41.20.1 `UnexpectedError.UnexpectedError()`

Member of `UnexpectedError`.

`UnexpectedError.UnexpectedError(String msg)`
`UnexpectedError.UnexpectedError(FusionException e)`

UnexpectedError.UnexpectedError(msg)

`msg : String`

UnexpectedError.UnexpectedError(e)

 e :   FusionException

5.41.21   fusion.UnimplementedError

Called a stub. Functionality has not yet been implemented.

 Constructors

       UnimplementedError(String msg)

5.41.21.1   UnimplementedError.UnimplementedError()

Member of UnimplementedError.
 UnimplementedError.UnimplementedError(String msg)

UnimplementedError.UnimplementedError(msg)

 msg :   String

5.41.22   fusion.ValueConversionError

Error casting or converting a value.

 Base class

       FusionRuntimeException

 Constructors

       ValueConversionError(String msg)

 Inherited methods

       FusionRuntimeException.toString()

5.41.22.1   ValueConversionError.ValueConversionError()

Member of ValueConversionError.
 ValueConversionError.ValueConversionError(String msg)

ValueConversionError.ValueConversionError(msg)

 msg :   String

5.42 Parameters

Solver parameter can be set using the Model.setSolverParam function.

Parameter value are either integers, floating point values or symbolic strings.

5.42.1 `allocAddQnz`

Additional number of Q non-zeros that are allocated space for when `numanz` exceeds `maxnumqnz` during addition of new Q entries.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

5000

Example:

```
M.setSolverParam("allocAddQnz", 5000)
```

5.42.2 `anaSolInfeasTol`

If a constraint violates its bound with an amount larger than this value, the constraint name, index and violation will be printed by the solution analyzer.

Accepted values:

Values in the range $[0.0; +\infty]$

Default value:

1e-6

Example:

```
M.setSolverParam("anaSolInfeasTol", 1e-6)
```

5.42.3 `autoSortABeforeOpt`

Controls whether the elements in each column of A are sorted before an optimization is performed. This is not required but makes the optimization more deterministic.

Accepted values:

One of the strings:

`on`
`off`

Default value:

```
off
```

Example:

```
M.setSolverParam("autoSortABeforeOpt", "off")
```

5.42.4 `autoUpdateSolInfo`

Controls whether the solution information items are automatically updated after an optimization is performed.

Accepted values:

One of the strings:
```
on
off
```

Default value:

```
off
```

Example:

```
M.setSolverParam("autoUpdateSolInfo", "off")
```

5.42.5 `basisRelTolS`

Maximum relative dual bound violation allowed in an optimal basic solution.

Accepted values:

Values in the range $[0.0; +\infty]$

Default value:

1.0e-12

Example:

```
M.setSolverParam("basisRelTolS", 1.0e-12)
```

5.42.6 `basisTolS`

Maximum absolute dual bound violation in an optimal basic solution.

Accepted values:

Values in the range $[1.0 \cdot 10^{-9}; +\infty]$

Default value:

    1.0e-6

Example:

```
M.setSolverParam("basisTolS", 1.0e-6)
```

5.42.7 `basisTolX`

Maximum absolute primal bound violation allowed in an optimal basic solution.

Accepted values:

    Values in the range $[1.0 \cdot 10^{-9}; +\infty]$

Default value:

    1.0e-6

Example:

```
M.setSolverParam("basisTolX", 1.0e-6)
```

5.42.8 `biCleanOptimizer`

Controls which simplex optimizer is used in the clean-up phase.

Accepted values:

    One of the strings:

```
intpnt
concurrent
mixedIntConic
mixedInt
dualSimplex
free
primalDualSimplex
conic
primalSimplex
networkPrimalSimplex
freeSimplex
```

Default value:

```
free
```

Example:

```
M.setSolverParam("biCleanOptimizer", "free")
```

5.42.9   `biIgnoreMaxIter`

If the parameter `intpntBasis` has the value "`noError`" and the interior-point optimizer has terminated due to maximum number of iterations, then basis identification is performed if this parameter has the value "`on`".

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
off
```

Example:

```
M.setSolverParam("biIgnoreMaxIter", "off")
```

5.42.10   `biIgnoreNumError`

If the parameter `intpntBasis` has the value "`noError`" and the interior-point optimizer has terminated due to a numerical problem, then basis identification is performed if this parameter has the value "`on`".

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
off
```

Example:

```
M.setSolverParam("biIgnoreNumError", "off")
```

5.42.11   `biMaxIterations`

Controls the maximum number of simplex iterations allowed to optimize a basis after the basis identification.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1000000

Example:

```
M.setSolverParam("biMaxIterations", 1000000)
```

5.42.12 `cacheLicense`

Specifies if the license is kept checked out for the lifetime of the mosek environment (on) or returned to the server immediately after the optimization (off).

By default the license is checked out for the lifetime of the MOSEK environment by the first call to "`optimizetrm`". The license is checked in when the environment is deleted.

A specific license feature may be checked in when not in use with the function "`checkinlicense`".

Check-in and check-out of licenses have an overhead. Frequent communication with the license server should be avoided.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("cacheLicense", "on")
```

5.42.13 `concurrentNumOptimizers`

The maximum number of simultaneous optimizations that will be started by the concurrent optimizer.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

2

Example:

```
M.setSolverParam("concurrentNumOptimizers", 2)
```

5.42.14 `concurrentPriorityDualSimplex`

Priority of the dual simplex algorithm when selecting solvers for concurrent optimization.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

2

Example:
```
M.setSolverParam("concurrentPriorityDualSimplex", 2)
```

5.42.15 concurrentPriorityFreeSimplex

Priority of the free simplex optimizer when selecting solvers for concurrent optimization.

Accepted values:
Integers in the range $[0; +\infty]$

Default value:
3

Example:
```
M.setSolverParam("concurrentPriorityFreeSimplex", 3)
```

5.42.16 concurrentPriorityIntpnt

Priority of the interior-point algorithm when selecting solvers for concurrent optimization.

Accepted values:
Integers in the range $[0; +\infty]$

Default value:
4

Example:
```
M.setSolverParam("concurrentPriorityIntpnt", 4)
```

5.42.17 concurrentPriorityPrimalSimplex

Priority of the primal simplex algorithm when selecting solvers for concurrent optimization.

Accepted values:
Integers in the range $[0; +\infty]$

Default value:
1

Example:
```
M.setSolverParam("concurrentPriorityPrimalSimplex", 1)
```

5.42.18 `infeasPreferPrimal`

If both certificates of primal and dual infeasibility are supplied then only the primal is used when this option is turned on.

Accepted values:

One of the strings:
```
on
off
```

Default value:
```
on
```

Example:
```
M.setSolverParam("infeasPreferPrimal", "on")
```

5.42.19 `intpntBasis`

Controls whether the interior-point optimizer also computes an optimal basis.

Accepted values:

One of the strings:
```
always
noError
never
ifFeasible
reservered
```

Default value:
```
always
```

Example:
```
M.setSolverParam("intpntBasis", "always")
```

See also:

- `biIgnoreMaxIter`
- `biIgnoreNumError`

5.42.20 `intpntCoTolDfeas`

Dual feasibility tolerance used by the conic interior-point optimizer.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

> 1.0e-8

Example:

> ```
> M.setSolverParam("intpntCoTolDfeas", 1.0e-8)
> ```

See also:

> - intpntCoTolNearRel

5.42.21   intpntCoTolInfeas

Controls when the conic interior-point optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Accepted values:

> Values in the range $[0.0; 1.0]$

Default value:

> 1.0e-10

Example:

> ```
> M.setSolverParam("intpntCoTolInfeas", 1.0e-10)
> ```

5.42.22   intpntCoTolMuRed

Relative complementarity gap tolerance feasibility tolerance used by the conic interior-point optimizer.

Accepted values:

> Values in the range $[0.0; 1.0]$

Default value:

> 1.0e-8

Example:

> ```
> M.setSolverParam("intpntCoTolMuRed", 1.0e-8)
> ```

5.42.23   intpntCoTolNearRel

If MOSEK cannot compute a solution that has the prescribed accuracy, then it will multiply the termination tolerances with value of this parameter. If the solution then satisfies the termination criteria, then the solution is denoted near optimal, near feasible and so forth.

Accepted values:

Values in the range $[1.0; +\infty]$

Default value:

1000

Example:

```
M.setSolverParam("intpntCoTolNearRel", 1000)
```

5.42.24 intpntCoTolPfeas

Primal feasibility tolerance used by the conic interior-point optimizer.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

1.0e-8

Example:

```
M.setSolverParam("intpntCoTolPfeas", 1.0e-8)
```

See also:

- intpntCoTolNearRel

5.42.25 intpntCoTolRelGap

Relative gap termination tolerance used by the conic interior-point optimizer.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

1.0e-7

Example:

```
M.setSolverParam("intpntCoTolRelGap", 1.0e-7)
```

See also:

- intpntCoTolNearRel

5.42.26   `intpntDiffStep`

Controls whether different step sizes are allowed in the primal and dual space.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("intpntDiffStep", "on")
```

5.42.27   `intpntFactorDebugLvl`

Controls factorization debug level.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

0

Example:

```
M.setSolverParam("intpntFactorDebugLvl", 0)
```

5.42.28   `intpntFactorMethod`

Controls the method used to factor the Newton equation system.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

0

Example:

```
M.setSolverParam("intpntFactorMethod", 0)
```

5.42.29 `intpntHotstart`

Currently not in use.

Accepted values:

One of the strings:

```
primal
none
dual
primalDual
```

Default value:

```
none
```

Example:

```
M.setSolverParam("intpntHotstart", "none")
```

5.42.30 `intpntMaxIterations`

Controls the maximum number of iterations allowed in the interior-point optimizer.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

400

Example:

```
M.setSolverParam("intpntMaxIterations", 400)
```

5.42.31 `intpntMaxNumCor`

Controls the maximum number of correctors allowed by the multiple corrector procedure. A negative value means that MOSEK is making the choice.

Accepted values:

Integers in the range $[-1; +\infty]$

Default value:

-1

Example:

```
M.setSolverParam("intpntMaxNumCor", -1)
```

5.42.32   `intpntMaxNumRefinementSteps`

Maximum number of steps to be used by the iterative refinement of the search direction. A negative value implies that the optimizer Chooses the maximum number of iterative refinement steps.

Accepted values:

Any integer value.

Default value:

-1

Example:

```
M.setSolverParam("intpntMaxNumRefinementSteps", -1)
```

5.42.33   `intpntOffColTrh`

Controls how many offending columns are detected in the Jacobian of the constraint matrix.

1 means aggressive detection, higher values mean less aggressive detection.

0 means no detection.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

40

Example:

```
M.setSolverParam("intpntOffColTrh", 40)
```

5.42.34   `intpntOrderMethod`

Controls the ordering strategy used by the interior-point optimizer when factorizing the Newton equation system.

Accepted values:

One of the strings:

```
none
forceGraphpar
appminloc
tryGraphpar
free
experimental
```

Default value:

```
free
```

Example:

```
M.setSolverParam("intpntOrderMethod", "free")
```

5.42.35 intpntRegularizationUse

Controls whether regularization is allowed.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("intpntRegularizationUse", "on")
```

5.42.36 intpntScaling

Controls how the problem is scaled before the interior-point optimizer is used.

Accepted values:

One of the strings:

```
none
moderate
aggressive
free
```

Default value:

```
free
```

Example:

```
M.setSolverParam("intpntScaling", "free")
```

5.42.37   `intpntSolveForm`

Controls whether the primal or the dual problem is solved.

Accepted values:

One of the strings:

```
primal
dual
free
```

Default value:

```
free
```

Example:

```
M.setSolverParam("intpntSolveForm", "free")
```

5.42.38   `intpntStartingPoint`

Starting point used by the interior-point optimizer.

Accepted values:

One of the strings:

```
guess
satisfyBounds
constant
free
```

Default value:

```
free
```

Example:

```
M.setSolverParam("intpntStartingPoint", "free")
```

5.42.39   `intpntTolDfeas`

Dual feasibility tolerance used for linear and quadratic optimization problems.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

1.0e-8

Example:

```
M.setSolverParam("intpntTolDfeas", 1.0e-8)
```

5.42.40 `intpntTolDsafe`

Controls the initial dual starting point used by the interior-point optimizer. If the interior-point optimizer converges slowly.

Accepted values:

Values in the range $[1.0 \cdot 10^{-4}; +\infty]$

Default value:

1.0

Example:

```
M.setSolverParam("intpntTolDsafe", 1.0)
```

5.42.41 `intpntTolInfeas`

Controls when the optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

1.0e-10

Example:

```
M.setSolverParam("intpntTolInfeas", 1.0e-10)
```

5.42.42 `intpntTolMuRed`

Relative complementarity gap tolerance.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

1.0e-16

Example:

```
M.setSolverParam("intpntTolMuRed", 1.0e-16)
```

5.42.43 `intpntTolPath`

Controls how close the interior-point optimizer follows the central path. A large value of this parameter means the central is followed very closely. On numerical unstable problems it may be worthwhile to increase this parameter.

Accepted values:

> Values in the range $[0.0; 0.9999]$

Default value:

> 1.0e-8

Example:

```
M.setSolverParam("intpntTolPath", 1.0e-8)
```

5.42.44 `intpntTolPfeas`

Primal feasibility tolerance used for linear and quadratic optimization problems.

Accepted values:

> Values in the range $[0.0; 1.0]$

Default value:

> 1.0e-8

Example:

```
M.setSolverParam("intpntTolPfeas", 1.0e-8)
```

5.42.45 `intpntTolPsafe`

Controls the initial primal starting point used by the interior-point optimizer. If the interior-point optimizer converges slowly and/or the constraint or variable bounds are very large, then it may be worthwhile to increase this value.

Accepted values:

> Values in the range $[1.0 \cdot 10^{-4}; +\infty]$

Default value:

> 1.0

Example:

```
M.setSolverParam("intpntTolPsafe", 1.0)
```

5.42.46 `intpntTolRelGap`

Relative gap termination tolerance.

Accepted values:

Values in the range $[1.0 \cdot 10^{-14}; +\infty]$

Default value:

1.0e-8

Example:

```
M.setSolverParam("intpntTolRelGap", 1.0e-8)
```

5.42.47 `intpntTolRelStep`

Relative step size to the boundary for linear and quadratic optimization problems.

Accepted values:

Values in the range $[1.0 \cdot 10^{-4}; 0.999999]$

Default value:

0.9999

Example:

```
M.setSolverParam("intpntTolRelStep", 0.9999)
```

5.42.48 `intpntTolStepSize`

If the step size falls below the value of this parameter, then the interior-point optimizer assumes that it is stalled. In other words the interior-point optimizer does not make any progress and therefore it is better stop.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

1.0e-6

Example:

```
M.setSolverParam("intpntTolStepSize", 1.0e-6)
```

5.42.49   licTrhExpiryWrn

If a license feature expires in a numbers days less than the value of this parameter then a warning will
be issued.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

7

Example:

```
M.setSolverParam("licTrhExpiryWrn", 7)
```

5.42.50   licenseDebug

This option is used to turn on debugging of the incense manager.

Accepted values:

One of the strings:

on
off

Default value:

off

Example:

```
M.setSolverParam("licenseDebug", "off")
```

5.42.51   licensePauseTime

If licenseWait="on" and no license is available, then MOSEK sleeps a number of milliseconds between
each check of whether a license has become free.

Accepted values:

Integers in the range $[0; 1000000]$

Default value:

100

Example:

```
M.setSolverParam("licensePauseTime", 100)
```

5.42.52 `licenseSuppressExpireWrns`

Controls whether license features expire warnings are suppressed.

Accepted values:

One of the strings:
```
on
off
```

Default value:

```
off
```

Example:
```
M.setSolverParam("licenseSuppressExpireWrns", "off")
```

5.42.53 `licenseWait`

If all licenses are in use MOSEK returns with an error code. However, by turning on this parameter MOSEK will wait for an available license.

Accepted values:

One of the strings:
```
on
off
```

Default value:

```
off
```

Example:
```
M.setSolverParam("licenseWait", "off")
```

5.42.54 `log`

Controls the amount of log information. The value 0 implies that all log information is suppressed. A higher level implies that more information is logged.

Please note that if a task is employed to solve a sequence of optimization problems the value of this parameter is reduced by the value of `logCutSecondOpt` for the second and any subsequent optimizations.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

10

Example:

```
M.setSolverParam("log", 10)
```

See also:

- logCutSecondOpt

5.42.55 logBi

Controls the amount of output printed by the basis identification procedure. A higher level implies that more information is logged.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

4

Example:

```
M.setSolverParam("logBi", 4)
```

5.42.56 logBiFreq

Controls how frequent the optimizer outputs information about the basis identification and how frequent the user-defined call-back function is called.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

2500

Example:

```
M.setSolverParam("logBiFreq", 2500)
```

5.42.57 logCheckConvexity

Controls logging in convexity check on quadratic problems. Set to a positive value to turn logging on.

If a quadratic coefficient matrix is found to violate the requirement of PSD (NSD) then a list of negative (positive) pivot elements is printed. The absolute value of the pivot elements is also shown.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

  0

Example:

```
M.setSolverParam("logCheckConvexity", 0)
```

5.42.58   `logConcurrent`

Controls amount of output printed by the concurrent optimizer.

  Accepted values:

  Integers in the range $[0; +\infty]$

  Default value:

  1

  Example:

```
M.setSolverParam("logConcurrent", 1)
```

5.42.59   `logCutSecondOpt`

If a task is employed to solve a sequence of optimization problems, then the value of the log levels is reduced by the value of this parameter. E.g `log` and `logSim` are reduced by the value of this parameter for the second and any subsequent optimizations.

  Accepted values:

  Integers in the range $[0; +\infty]$

  Default value:

  1

  Example:

```
M.setSolverParam("logCutSecondOpt", 1)
```

  See also:

  - `log`
  - `logIntpnt`
  - `logMio`
  - `logSim`

5.42.60   logExpand

Controls the amount of logging when a data item such as the maximum number constrains is expanded.

Accepted values:

>   Integers in the range $[0; +\infty]$

Default value:

>   0

Example:

```
M.setSolverParam("logExpand", 0)
```

5.42.61   logFactor

If turned on, then the factor log lines are added to the log.

Accepted values:

>   Integers in the range $[0; +\infty]$

Default value:

>   1

Example:

```
M.setSolverParam("logFactor", 1)
```

5.42.62   logFeasRepair

Controls the amount of output printed when performing feasibility repair. A value higher than one means extensive logging.

Accepted values:

>   Integers in the range $[0; +\infty]$

Default value:

>   1

Example:

```
M.setSolverParam("logFeasRepair", 1)
```

5.42.63 `logFile`

If turned on, then some log info is printed when a file is written or read.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1

Example:

```
M.setSolverParam("logFile", 1)
```

5.42.64 `logHead`

If turned on, then a header line is added to the log.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1

Example:

```
M.setSolverParam("logHead", 1)
```

5.42.65 `logInfeasAna`

Controls amount of output printed by the infeasibility analyzer procedures. A higher level implies that more information is logged.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1

Example:

```
M.setSolverParam("logInfeasAna", 1)
```

5.42.66 `logIntpnt`

Controls amount of output printed printed by the interior-point optimizer. A higher level implies that more information is logged.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

4

Example:

```
M.setSolverParam("logIntpnt", 4)
```

5.42.67 `logMio`

Controls the log level for the mixed-integer optimizer. A higher level implies that more information is logged.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

4

Example:

```
M.setSolverParam("logMio", 4)
```

5.42.68 `logMioFreq`

Controls how frequent the mixed-integer optimizer prints the log line. It will print line every time `logMioFreq` relaxations have been solved.

Accepted values:

Any integer value.

Default value:

1000

Example:

```
M.setSolverParam("logMioFreq", 1000)
```

5.42.69 `logNonconvex`

Controls amount of output printed by the nonconvex optimizer.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1

Example:

```
M.setSolverParam("logNonconvex", 1)
```

5.42.70 `logOptimizer`

Controls the amount of general optimizer information that is logged.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1

Example:

```
M.setSolverParam("logOptimizer", 1)
```

5.42.71 `logOrder`

If turned on, then factor lines are added to the log.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1

Example:

```
M.setSolverParam("logOrder", 1)
```

5.42.72 logParam

Controls the amount of information printed out about parameter changes.

Accepted values:

   Integers in the range $[0; +\infty]$

Default value:

   0

Example:

```
M.setSolverParam("logParam", 0)
```

5.42.73 logPresolve

Controls amount of output printed by the presolve procedure. A higher level implies that more information is logged.

Accepted values:

   Integers in the range $[0; +\infty]$

Default value:

   1

Example:

```
M.setSolverParam("logPresolve", 1)
```

5.42.74 logResponse

Controls amount of output printed when response codes are reported. A higher level implies that more information is logged.

Accepted values:

   Integers in the range $[0; +\infty]$

Default value:

   0

Example:

```
M.setSolverParam("logResponse", 0)
```

5.42.75 `logSim`

Controls amount of output printed by the simplex optimizer. A higher level implies that more information is logged.

Accepted values:

> Integers in the range $[0; +\infty]$

Default value:

> 4

Example:

```
M.setSolverParam("logSim", 4)
```

5.42.76 `logSimFreq`

Controls how frequent the simplex optimizer outputs information about the optimization and how frequent the user-defined call-back function is called.

Accepted values:

> Integers in the range $[0; +\infty]$

Default value:

> 1000

Example:

```
M.setSolverParam("logSimFreq", 1000)
```

5.42.77 `logSimMinor`

Currently not in use.

Accepted values:

> Integers in the range $[0; +\infty]$

Default value:

> 1

Example:

```
M.setSolverParam("logSimMinor", 1)
```

5.42.78   logSimNetworkFreq

Controls how frequent the network simplex optimizer outputs information about the optimization and how frequent the user-defined call-back function is called. The network optimizer will use a logging frequency equal to logSimFreq times logSimNetworkFreq.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1000

Example:

```
M.setSolverParam("logSimNetworkFreq", 1000)
```

5.42.79   logStorage

When turned on, MOSEK prints messages regarding the storage usage and allocation.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

0

Example:

```
M.setSolverParam("logStorage", 0)
```

5.42.80   lowerObjCut

If either a primal or dual feasible solution is found proving that the optimal objective value is outside, the interval [lowerObjCut, upperObjCut], then MOSEK is terminated.

Accepted values:

Any value.

Default value:

-1.0e30

Example:

```
M.setSolverParam("lowerObjCut", -1.0e30)
```

See also:

- lowerObjCutFiniteTrh

5.42.81 `lowerObjCutFiniteTrh`

If the lower objective cut is less than the value of this parameter value, then the lower objective cut i.e. `lowerObjCut` is treated as $-\infty$.

Accepted values:

   Any value.

Default value:

   -0.5e30

Example:

```
M.setSolverParam("lowerObjCutFiniteTrh", -0.5e30)
```

5.42.82 `maxNumWarnings`

A negtive number means all warnings are logged. Otherwise the parameter specifies the maximum number times each warning is logged.

Accepted values:

   Any integer value.

Default value:

   6

Example:

```
M.setSolverParam("maxNumWarnings", 6)
```

5.42.83 `mioBranchDir`

Controls whether the mixed-integer optimizer is branching up or down by default.

Accepted values:

   One of the strings:

   ```
   down
   up
   free
   ```

Default value:

   ```
   free
   ```

Example:

```
M.setSolverParam("mioBranchDir", "free")
```

5.42.84   mioBranchPrioritiesUse

Controls whether branching priorities are used by the mixed-integer optimizer.

Accepted values:

One of the strings:
```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("mioBranchPrioritiesUse", "on")
```

5.42.85   mioConstructSol

If set to "on" and all integer variables have been given a value for which a feasible mixed integer solution exists, then MOSEK generates an initial solution to the mixed integer problem by fixing all integer values and solving the remaining problem.

Accepted values:

One of the strings:
```
on
off
```

Default value:

```
off
```

Example:

```
M.setSolverParam("mioConstructSol", "off")
```

5.42.86   mioContSol

Controls the meaning of the interior-point and basic solutions in mixed integer problems.

Accepted values:

One of the strings:
```
itg
none
root
itgRel
```

Default value:

```
none
```

Example:

```
M.setSolverParam("mioContSol", "none")
```

5.42.87 mioCutCg

Controls whether CG cuts should be generated.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("mioCutCg", "on")
```

5.42.88 mioCutCmir

Controls whether mixed integer rounding cuts should be generated.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("mioCutCmir", "on")
```

5.42.89 mioCutLevelRoot

Controls the cut level employed by the mixed-integer optimizer at the root node. A negative value means a default value determined by the mixed-integer optimizer is used. By adding the appropriate values from the following table the employed cut types can be controlled.

GUB cover	+2
Flow cover	+4
Lifting	+8
Plant location	+16
Disaggregation	+32
Knapsack cover	+64
Lattice	+128
Gomory	+256
Coefficient reduction	+512
GCD	+1024
Obj. integrality	+2048

Accepted values:

   Any integer value.

Default value:

   -1

Example:

```
M.setSolverParam("mioCutLevelRoot", -1)
```

5.42.90   mioCutLevelTree

Controls the cut level employed by the mixed-integer optimizer at the tree. See `mioCutLevelRoot` for an explanation of the parameter values.

Accepted values:

   Any integer value.

Default value:

   -1

Example:

```
M.setSolverParam("mioCutLevelTree", -1)
```

5.42.91   mioDisableTermTime

The termination criteria governed by

- `mioMaxNumRelaxs`

- `mioMaxNumBranches`

- `mioNearTolAbsGap`

- `mioNearTolRelGap`

is disabled the first n seconds. This parameter specifies the number n. A negative value is identical to infinity i.e. the termination criteria are never checked.

Accepted values:

Any value.

Default value:

-1.0

Example:

```
M.setSolverParam("mioDisableTermTime", -1.0)
```

See also:

- `mioMaxNumRelaxs`
- `mioMaxNumBranches`
- `mioNearTolAbsGap`
- `mioNearTolRelGap`

5.42.92 `mioFeaspumpLevel`

Feasibility pump is a heuristic designed to compute an initial feasible solution. A value of 0 implies that the feasibility pump heuristic is not used. A value of -1 implies that the mixed-integer optimizer decides how the feasibility pump heuristic is used. A larger value than 1 implies that the feasibility pump is employed more aggressively. Normally a value beyond 3 is not worthwhile.

Accepted values:

Integers in the range $[-\infty; 3]$

Default value:

-1

Example:

```
M.setSolverParam("mioFeaspumpLevel", -1)
```

5.42.93 `mioHeuristicLevel`

Controls the heuristic employed by the mixed-integer optimizer to locate an initial good integer feasible solution. A value of zero means the heuristic is not used at all. A larger value than 0 means that a gradually more sophisticated heuristic is used which is computationally more expensive. A negative value implies that the optimizer chooses the heuristic. Normally a value around 3 to 5 should be optimal.

Accepted values:

Any integer value.

Default value:

-1

Example:

```
M.setSolverParam("mioHeuristicLevel", -1)
```

5.42.94 mioHeuristicTime

Minimum amount of time to be used in the heuristic search for a good feasible integer solution. A negative values implies that the optimizer decides the amount of time to be spent in the heuristic.

Accepted values:

Any value.

Default value:

-1.0

Example:

```
M.setSolverParam("mioHeuristicTime", -1.0)
```

5.42.95 mioHotstart

Controls whether the integer optimizer is hot-started.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("mioHotstart", "on")
```

5.42.96 `mioKeepBasis`

Controls whether the integer presolve keeps bases in memory. This speeds on the solution process at cost of bigger memory consumption.

Accepted values:

   One of the strings:
   ```
   on
   off
   ```

Default value:

   ```
   on
   ```

Example:

   ```
   M.setSolverParam("mioKeepBasis", "on")
   ```

5.42.97 `mioLocalBranchNumber`

Controls the size of the local search space when doing local branching.

Accepted values:

   Any integer value.

Default value:

   -1

Example:

   ```
   M.setSolverParam("mioLocalBranchNumber", -1)
   ```

5.42.98 `mioMaxNumBranches`

Maximum number of branches allowed during the branch and bound search. A negative value means infinite.

Accepted values:

   Any integer value.

Default value:

   -1

Example:

   ```
   M.setSolverParam("mioMaxNumBranches", -1)
   ```

See also:

   - mioDisableTermTime

5.42.99 mioMaxNumRelaxs

Maximum number of relaxations allowed during the branch and bound search. A negative value means infinite.

Accepted values:

   Any integer value.

Default value:

   -1

Example:

```
M.setSolverParam("mioMaxNumRelaxs", -1)
```

See also:

   • mioDisableTermTime

5.42.100 mioMaxNumSolutions

The mixed-integer optimizer can be terminated after a certain number of different feasible solutions has been located. If this parameter has the value n and n is strictly positive, then the mixed-integer optimizer will be terminated when n feasible solutions have been located.

Accepted values:

   Any integer value.

Default value:

   -1

Example:

```
M.setSolverParam("mioMaxNumSolutions", -1)
```

See also:

   • mioDisableTermTime

5.42.101 mioMaxTime

This parameter limits the maximum time spent by the mixed-integer optimizer. A negative number means infinity.

Accepted values:

   Any value.

Default value:

    -1.0

Example:

    `M.setSolverParam("mioMaxTime", -1.0)`

5.42.102 `mioMaxTimeAprxOpt`

Number of seconds spent by the mixed-integer optimizer before the `mioTolRelRelaxInt` is applied.

Accepted values:

    Values in the range $[0.0; +\infty]$

Default value:

    60

Example:

    `M.setSolverParam("mioMaxTimeAprxOpt", 60)`

5.42.103 `mioMode`

Controls whether the optimizer includes the integer restrictions when solving a (mixed) integer optimization problem.

Accepted values:

    One of the strings:

        `ignored`
        `lazy`
        `satisfied`

Default value:

        `satisfied`

Example:

    `M.setSolverParam("mioMode", "satisfied")`

5.42.104 `mioMtUserCb`

It true user callbacks are called from each thread used by this optimizer. If false the user callback is only called from a single thread.

Accepted values:

> One of the strings:
> ```
> on
> off
> ```

Default value:

> ```
> on
> ```

Example:

> ```
> M.setSolverParam("mioMtUserCb", "on")
> ```

5.42.105   `mioNearTolAbsGap`

Relaxed absolute optimality tolerance employed by the mixed-integer optimizer. This termination criteria is delayed. See `mioDisableTermTime` for details.

Accepted values:

> Values in the range $[0.0; +\infty]$

Default value:

> 0.0

Example:

> ```
> M.setSolverParam("mioNearTolAbsGap", 0.0)
> ```

See also:

> • `mioDisableTermTime`

5.42.106   `mioNearTolRelGap`

The mixed-integer optimizer is terminated when this tolerance is satisfied. This termination criteria is delayed. See `mioDisableTermTime` for details.

Accepted values:

> Values in the range $[0.0; +\infty]$

Default value:

> 1.0e-3

Example:

> ```
> M.setSolverParam("mioNearTolRelGap", 1.0e-3)
> ```

See also:

> • `mioDisableTermTime`

5.42.107 `mioNodeOptimizer`

Controls which optimizer is employed at the non-root nodes in the mixed-integer optimizer.

Accepted values:

One of the strings:

```
intpnt
concurrent
mixedIntConic
mixedInt
dualSimplex
free
primalDualSimplex
conic
primalSimplex
networkPrimalSimplex
freeSimplex
```

Default value:

```
free
```

Example:

```
M.setSolverParam("mioNodeOptimizer", "free")
```

5.42.108 `mioNodeSelection`

Controls the node selection strategy employed by the mixed-integer optimizer.

Accepted values:

One of the strings:

```
pseudo
hybrid
free
worst
best
first
```

Default value:

```
free
```

Example:

```
M.setSolverParam("mioNodeSelection", "free")
```

5.42.109 `mioOptimizerMode`

An exprimental feature.

Accepted values:

> Integers in the range $[0; 1]$

Default value:

> 0

Example:

```
M.setSolverParam("mioOptimizerMode", 0)
```

5.42.110 `mioPresolveAggregate`

Controls whether the presolve used by the mixed-integer optimizer tries to aggregate the constraints.

Accepted values:

> One of the strings:
>
> on
> off

Default value:

> on

Example:

```
M.setSolverParam("mioPresolveAggregate", "on")
```

5.42.111 `mioPresolveProbing`

Controls whether the mixed-integer presolve performs probing. Probing can be very time consuming.

Accepted values:

> One of the strings:
>
> on
> off

Default value:

> on

Example:

```
M.setSolverParam("mioPresolveProbing", "on")
```

5.42.112 `mioPresolveUse`

Controls whether presolve is performed by the mixed-integer optimizer.

Accepted values:
>   One of the strings:
>   ```
>   on
>   off
>   ```

Default value:
>   ```
>   on
>   ```

Example:
>   ```
>   M.setSolverParam("mioPresolveUse", "on")
>   ```

5.42.113 `mioProbingLevel`

Controls the amount of probing employed by the mixed-integer optimizer in presolve.

- -1 The optimizer chooses the level of probing employed.

- 0 Probing is disabled.

- 1 A low amount of probing is employed.

- 2 A medium amount of probing is employed.

- 3 A high amount of probing is employed.

Accepted values:
>   Any integer value.

Default value:
>   -1

Example:
>   ```
>   M.setSolverParam("mioProbingLevel", -1)
>   ```

5.42.114 `mioRelAddCutLimited`

Controls how many cuts the mixed-integer optimizer is allowed to add to the problem. Let α be the value of this parameter and m the number constraints, then mixed-integer optimizer is allowed to αm cuts.

Accepted values:
>   Values in the range $[0.0; 2.0]$

Default value:

    0.75

Example:

```
M.setSolverParam("mioRelAddCutLimited", 0.75)
```

5.42.115 mioRelGapConst

This value is used to compute the relative gap for the solution to an integer optimization problem.

Accepted values:

    Values in the range $[1.0 \cdot 10^{-15}; +\infty]$

Default value:

    1.0e-10

Example:

```
M.setSolverParam("mioRelGapConst", 1.0e-10)
```

5.42.116 mioRinsMaxNodes

Controls the maximum number of nodes allowed in each call to the RINS heuristic. The default value of -1 means that the value is determined automatically. A value of zero turns off the heuristic.

Accepted values:

    Integers in the range $[-1; +\infty]$

Default value:

    -1

Example:

```
M.setSolverParam("mioRinsMaxNodes", -1)
```

5.42.117 mioRootOptimizer

Controls which optimizer is employed at the root node in the mixed-integer optimizer.

Accepted values:

    One of the strings:

    ```
    intpnt
    concurrent
    mixedIntConic
    mixedInt
    ```

```
dualSimplex
free
primalDualSimplex
conic
primalSimplex
networkPrimalSimplex
freeSimplex
```

Default value:

```
free
```

Example:

```
M.setSolverParam("mioRootOptimizer", "free")
```

5.42.118 mioStrongBranch

The value specifies the depth from the root in which strong branching is used. A negative value means that the optimizer chooses a default value automatically.

Accepted values:

Any integer value.

Default value:

-1

Example:

```
M.setSolverParam("mioStrongBranch", -1)
```

5.42.119 mioTolAbsGap

Absolute optimality tolerance employed by the mixed-integer optimizer.

Accepted values:

Values in the range $[0.0; +\infty]$

Default value:

0.0

Example:

```
M.setSolverParam("mioTolAbsGap", 0.0)
```

5.42.120   `mioTolAbsRelaxInt`

Absolute relaxation tolerance of the integer constraints. I.e. $\min(|x| - \lfloor x \rfloor, \lceil x \rceil - |x|)$ is less than the tolerance then the integer restrictions assumed to be satisfied.

Accepted values:

   Values in the range $[1 \cdot 10^{-9}; +\infty]$

Default value:

   1.0e-5

Example:

```
M.setSolverParam("mioTolAbsRelaxInt", 1.0e-5)
```

5.42.121   `mioTolFeas`

Feasibility tolerance for mixed integer solver. Any solution with maximum infeasibility below this value will be considered feasible.

Accepted values:

   Values in the range $[0.0; +\infty]$

Default value:

   1.0e-7

Example:

```
M.setSolverParam("mioTolFeas", 1.0e-7)
```

5.42.122   `mioTolMaxCutFracRhs`

Maximum value of fractional part of right hand side to generate CMIR and CG cuts for. A value of 0.0 means that the value is selected automatically.

Accepted values:

   Values in the range $[0.0; 1.0]$

Default value:

   0.0

Example:

```
M.setSolverParam("mioTolMaxCutFracRhs", 0.0)
```

5.42.123 `mioTolMinCutFracRhs`

Minimum value of fractional part of right hand side to generate CMIR and CG cuts for. A value of 0.0 means that the value is selected automatically.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

0.0

Example:

```
M.setSolverParam("mioTolMinCutFracRhs", 0.0)
```

5.42.124 `mioTolRelDualBoundImprovement`

If the relative improvement of the dual bound is smaller than this value, the solver will terminate the root cut generation. A value of 0.0 means that the value is selected automatically.

Accepted values:

Values in the range $[0.0; 1.0]$

Default value:

0.0

Example:

```
M.setSolverParam("mioTolRelDualBoundImprovement", 0.0)
```

5.42.125 `mioTolRelGap`

Relative optimality tolerance employed by the mixed-integer optimizer.

Accepted values:

Values in the range $[0.0; +\infty]$

Default value:

1.0e-4

Example:

```
M.setSolverParam("mioTolRelGap", 1.0e-4)
```

5.42.126 `mioTolRelRelaxInt`

Relative relaxation tolerance of the integer constraints. I.e $(\min(|x| - \lfloor x \rfloor, \lceil x \rceil - |x|))$ is less than the tolerance times $|x|$ then the integer restrictions assumed to be satisfied.

Accepted values:

   Values in the range $[0.0; +\infty]$

Default value:

   1.0e-6

Example:

   `M.setSolverParam("mioTolRelRelaxInt", 1.0e-6)`

5.42.127 `mioTolX`

Absolute solution tolerance used in mixed-integer optimizer.

Accepted values:

   Values in the range $[0.0; +\infty]$

Default value:

   1.0e-6

Example:

   `M.setSolverParam("mioTolX", 1.0e-6)`

5.42.128 `mioUseMultithreadedOptimizer`

Controls wheter the new multithreaded optimizer should be used for Mixed integer problems.

Accepted values:

   One of the strings:

   `on`
   `off`

Default value:

   `off`

Example:

   `M.setSolverParam("mioUseMultithreadedOptimizer", "off")`

5.42.129 `mtSpincount`

Set the number of iterations to spin before sleeping.

Accepted values:

Integers in the range $[0; 1000000000]$

Default value:

0

Example:

```
M.setSolverParam("mtSpincount", 0)
```

5.42.130 `numThreads`

Controls the number of threads employed by the optimizer. If set to 0 the number of threads used will be equal to the number of cores detected on the machine.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

0

Example:

```
M.setSolverParam("numThreads", 0)
```

5.42.131 `optimizer`

The paramter controls which optimizer is used to optimize the task.

Accepted values:

One of the strings:

```
intpnt
concurrent
mixedIntConic
mixedInt
dualSimplex
free
primalDualSimplex
conic
primalSimplex
networkPrimalSimplex
freeSimplex
```

Default value:

> ```
> free
> ```

Example:

> ```
> M.setSolverParam("optimizer", "free")
> ```

5.42.132   optimizerMaxTime

Maximum amount of time the optimizer is allowed to spent on the optimization. A negative number means infinity.

Accepted values:

> Any value.

Default value:

> -1.0

Example:

> ```
> M.setSolverParam("optimizerMaxTime", -1.0)
> ```

5.42.133   presolveElimFill

Controls the maximum amount of fill-in that can be created during the elimination phase of the presolve. This parameter times (numcon+numvar) denotes the amount of fill-in.

Accepted values:

> Integers in the range $[0; +\infty]$

Default value:

> 1

Example:

> ```
> M.setSolverParam("presolveElimFill", 1)
> ```

5.42.134   presolveEliminatorMaxNumTries

Control the maximum number of times the eliminator is tried.

Accepted values:

> Any integer value.

Default value:

> -1

Example:

```
M.setSolverParam("presolveEliminatorMaxNumTries", -1)
```

5.42.135 `presolveEliminatorUse`

Controls whether free or implied free variables are eliminated from the problem.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("presolveEliminatorUse", "on")
```

5.42.136 `presolveLevel`

Currently not used.

Accepted values:

Any integer value.

Default value:

-1

Example:

```
M.setSolverParam("presolveLevel", -1)
```

5.42.137 `presolveLindepAbsWorkTrh`

The linear dependency check is potentially computationally expensive.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

100

Example:

```
M.setSolverParam("presolveLindepAbsWorkTrh", 100)
```

5.42.138   `presolveLindepRelWorkTrh`

The linear dependency check is potentially computationally expensive.

Accepted values:

> Integers in the range $[0; +\infty]$

Default value:

> 100

Example:

```
M.setSolverParam("presolveLindepRelWorkTrh", 100)
```

5.42.139   `presolveLindepUse`

Controls whether the linear constraints are checked for linear dependencies.

Accepted values:

> One of the strings:
> ```
> on
> off
> ```

Default value:

> ```
> on
> ```

Example:

```
M.setSolverParam("presolveLindepUse", "on")
```

5.42.140   `presolveMaxNumReductions`

Controls the maximum number reductions performed by the presolve. The value of the parameter is normally only changed in connection with debugging. A negative value implies that an infinite number of reductions are allowed.

Accepted values:

> Any integer value.

Default value:

> -1

Example:

```
M.setSolverParam("presolveMaxNumReductions", -1)
```

5.42.141 `presolveTolAbsLindep`

Absolute tolerance employed by the linear dependency checker.

Accepted values:

Values in the range $[0.0; +\infty]$

Default value:

1.0e-6

Example:

```
M.setSolverParam("presolveTolAbsLindep", 1.0e-6)
```

5.42.142 `presolveTolAij`

Absolute zero tolerance employed for a_{ij} in the presolve.

Accepted values:

Values in the range $[1.0 \cdot 10^{-15}; +\infty]$

Default value:

1.0e-12

Example:

```
M.setSolverParam("presolveTolAij", 1.0e-12)
```

5.42.143 `presolveTolRelLindep`

Relative tolerance employed by the linear dependency checker.

Accepted values:

Values in the range $[0.0; +\infty]$

Default value:

1.0e-10

Example:

```
M.setSolverParam("presolveTolRelLindep", 1.0e-10)
```

5.42.144   presolveTolS

Absolute zero tolerance employed for s_i in the presolve.

Accepted values:

Values in the range $[0.0; +\infty]$

Default value:

1.0e-8

Example:

```
M.setSolverParam("presolveTolS", 1.0e-8)
```

5.42.145   presolveTolX

Absolute zero tolerance employed for x_j in the presolve.

Accepted values:

Values in the range $[0.0; +\infty]$

Default value:

1.0e-8

Example:

```
M.setSolverParam("presolveTolX", 1.0e-8)
```

5.42.146   presolveUse

Controls whether the presolve is applied to a problem before it is optimized.

Accepted values:

One of the strings:

```
on
off
free
```

Default value:

```
free
```

Example:

```
M.setSolverParam("presolveUse", "free")
```

5.42.147 `simBasisFactorUse`

Controls whether a (LU) factorization of the basis is used in a hot-start. Forcing a refactorization sometimes improves the stability of the simplex optimizers, but in most cases there is a performance penanlty.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
on
```

Example:

```
M.setSolverParam("simBasisFactorUse", "on")
```

5.42.148 `simDegen`

Controls how aggressively degeneration is handled.

Accepted values:

One of the strings:

```
none
moderate
minimum
aggressive
free
```

Default value:

```
free
```

Example:

```
M.setSolverParam("simDegen", "free")
```

5.42.149 `simDualCrash`

Controls whether crashing is performed in the dual simplex optimizer.

In general if a basis consists of more than (100-this parameter value)% fixed variables, then a crash will be performed.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

> 90

Example:

```
M.setSolverParam("simDualCrash", 90)
```

5.42.150   simDualPhaseoneMethod

An exprimental feature.

Accepted values:

> Integers in the range $[0; 10]$

Default value:

> 0

Example:

```
M.setSolverParam("simDualPhaseoneMethod", 0)
```

5.42.151   simDualRestrictSelection

The dual simplex optimizer can use a so-called restricted selection/pricing strategy to chooses the outgoing variable. Hence, if restricted selection is applied, then the dual simplex optimizer first choose a subset of all the potential outgoing variables. Next, for some time it will choose the outgoing variable only among the subset. From time to time the subset is redefined.

A larger value of this parameter implies that the optimizer will be more aggressive in its restriction strategy, i.e. a value of 0 implies that the restriction strategy is not applied at all.

Accepted values:

> Integers in the range $[0; 100]$

Default value:

> 50

Example:

```
M.setSolverParam("simDualRestrictSelection", 50)
```

5.42.152   simDualSelection

Controls the choice of the incoming variable, known as the selection strategy, in the dual simplex optimizer.

Accepted values:

One of the strings:

```
full
partial
free
ase
devex
se
```

Default value:

```
free
```

Example:

```
M.setSolverParam("simDualSelection", "free")
```

5.42.153 simExploitDupvec

Controls if the simplex optimizers are allowed to exploit duplicated columns.

Accepted values:

One of the strings:

```
on
off
free
```

Default value:

```
off
```

Example:

```
M.setSolverParam("simExploitDupvec", "off")
```

5.42.154 simHotstart

Controls the type of hot-start that the simplex optimizer perform.

Accepted values:

One of the strings:

```
none
statusKeys
free
```

Default value:

```
free
```

Example:

```
M.setSolverParam("simHotstart", "free")
```

5.42.155   `simHotstartLu`

Determines if the simplex optimizer should exploit the initial factorization.

   Accepted values:

    One of the strings:

      `on`
      `off`

   Default value:

      `on`

   Example:

```
M.setSolverParam("simHotstartLu", "on")
```

5.42.156   `simInteger`

An exprimental feature.

   Accepted values:

    Integers in the range $[0; 10]$

   Default value:

    0

   Example:

```
M.setSolverParam("simInteger", 0)
```

5.42.157   `simLuTolRelPiv`

Relative pivot tolerance employed when computing the LU factorization of the basis in the simplex optimizers and in the basis identification procedure.

A value closer to 1.0 generally improves numerical stability but typically also implies an increase in the computational work.

   Accepted values:

    Values in the range $[1.0 \cdot 10^{-6}; 0.999999]$

   Default value:

    0.01

   Example:

```
M.setSolverParam("simLuTolRelPiv", 0.01)
```

5.42.158 `simMaxIterations`

Maximum number of iterations that can be used by a simplex optimizer.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

10000000

Example:

```
M.setSolverParam("simMaxIterations", 10000000)
```

5.42.159 `simMaxNumSetbacks`

Controls how many set-backs are allowed within a simplex optimizer. A set-back is an event where the optimizer moves in the wrong direction. This is impossible in theory but may happen due to numerical problems.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

250

Example:

```
M.setSolverParam("simMaxNumSetbacks", 250)
```

5.42.160 `simNonSingular`

Controls if the simplex optimizer ensures a non-singular basis, if possible.

Accepted values:

One of the strings:

on
off

Default value:

on

Example:

```
M.setSolverParam("simNonSingular", "on")
```

5.42.161   `simPrimalCrash`

Controls whether crashing is performed in the primal simplex optimizer.

In general, if a basis consists of more than (100-this parameter value)% fixed variables, then a crash will be performed.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

90

Example:

```
M.setSolverParam("simPrimalCrash", 90)
```

5.42.162   `simPrimalPhaseoneMethod`

An exprimental feature.

Accepted values:

Integers in the range $[0; 10]$

Default value:

0

Example:

```
M.setSolverParam("simPrimalPhaseoneMethod", 0)
```

5.42.163   `simPrimalRestrictSelection`

The primal simplex optimizer can use a so-called restricted selection/pricing strategy to chooses the outgoing variable. Hence, if restricted selection is applied, then the primal simplex optimizer first choose a subset of all the potential incoming variables. Next, for some time it will choose the incoming variable only among the subset. From time to time the subset is redefined.

A larger value of this parameter implies that the optimizer will be more aggressive in its restriction strategy, i.e. a value of 0 implies that the restriction strategy is not applied at all.

Accepted values:

Integers in the range $[0; 100]$

Default value:

50

Example:

```
M.setSolverParam("simPrimalRestrictSelection", 50)
```

5.42.164 `simPrimalSelection`

Controls the choice of the incoming variable, known as the selection strategy, in the primal simplex optimizer.

Accepted values:

One of the strings:

```
full
partial
free
ase
devex
se
```

Default value:

```
free
```

Example:

```
M.setSolverParam("simPrimalSelection", "free")
```

5.42.165 `simRefactorFreq`

Controls how frequent the basis is refactorized. The value 0 means that the optimizer determines the best point of refactorization.

It is strongly recommended NOT to change this parameter.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

0

Example:

```
M.setSolverParam("simRefactorFreq", 0)
```

5.42.166 `simReformulation`

Controls if the simplex optimizers are allowed to reformulate the problem.

Accepted values:

One of the strings:

```
on
aggressive
off
free
```

Default value:

```
off
```

Example:

```
M.setSolverParam("simReformulation", "off")
```

5.42.167   simSaveLu

Controls if the LU factorization stored should be replaced with the LU factorization corresponding to the initial basis.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
off
```

Example:

```
M.setSolverParam("simSaveLu", "off")
```

5.42.168   simScaling

Controls how much effort is used in scaling the problem before a simplex optimizer is used.

Accepted values:

One of the strings:

```
none
moderate
aggressive
free
```

Default value:

```
free
```

Example:

```
M.setSolverParam("simScaling", "free")
```

5.42.169 `simScalingMethod`

Controls how the problem is scaled before a simplex optimizer is used.

Accepted values:

One of the strings:

`pow2`
`free`

Default value:

`pow2`

Example:

`M.setSolverParam("simScalingMethod", "pow2")`

5.42.170 `simSolveForm`

Controls whether the primal or the dual problem is solved by the primal-/dual- simplex optimizer.

Accepted values:

One of the strings:

`primal`
`dual`
`free`

Default value:

`free`

Example:

`M.setSolverParam("simSolveForm", "free")`

5.42.171 `simStabilityPriority`

Controls how high priority the numerical stability should be given.

Accepted values:

Integers in the range $[0; 100]$

Default value:

50

Example:

`M.setSolverParam("simStabilityPriority", 50)`

5.42.172 `simSwitchOptimizer`

The simplex optimizer sometimes chooses to solve the dual problem instead of the primal problem. This implies that if you have chosen to use the dual simplex optimizer and the problem is dualized, then it actually makes sense to use the primal simplex optimizer instead. If this parameter is on and the problem is dualized and furthermore the simplex optimizer is chosen to be the primal (dual) one, then it is switched to the dual (primal).

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
off
```

Example:

```
M.setSolverParam("simSwitchOptimizer", "off")
```

5.42.173 `simplexAbsTolPiv`

Absolute pivot tolerance employed by the simplex optimizers.

Accepted values:

Values in the range $[1.0 \cdot 10^{-12}; +\infty]$

Default value:

1.0e-7

Example:

```
M.setSolverParam("simplexAbsTolPiv", 1.0e-7)
```

5.42.174 `solFilterKeepBasic`

If turned on, then basic and super basic constraints and variables are written to the solution file independent of the filter setting.

Accepted values:

One of the strings:

```
on
off
```

Default value:

```
off
```

Example:

```
M.setSolverParam("solFilterKeepBasic", "off")
```

5.42.175 `timingLevel`

Controls the a amount of timing performed inside MOSEK.

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1

Example:

```
M.setSolverParam("timingLevel", 1)
```

5.42.176 `upperObjCut`

If either a primal or dual feasible solution is found proving that the optimal objective value is outside, [`lowerObjCut`, `upperObjCut`], then MOSEK is terminated.

Accepted values:

Any value.

Default value:

1.0e30

Example:

```
M.setSolverParam("upperObjCut", 1.0e30)
```

See also:

- `upperObjCutFiniteTrh`

5.42.177 `upperObjCutFiniteTrh`

If the upper objective cut is greater than the value of this value parameter, then the the upper objective cut `upperObjCut` is treated as ∞.

Accepted values:

Any value.

Default value:

0.5e30

Example:

```
M.setSolverParam("upperObjCutFiniteTrh", 0.5e30)
```

5.42.178 warningLevel

Deprecated and not in use

Accepted values:

Integers in the range $[0; +\infty]$

Default value:

1

Example:

```
M.setSolverParam("warningLevel", 1)
```

5.43   Information items

Solver nformation items can be extracted using the Model.getSolverDoubleInfo, Model.getSolverIntInfo and Model.getSolverLIntInfo functions.

Information item values are either floating point, integer or long integer values.

5.43.1   Double information items

biCleanDualTime

Time spent within the dual clean-up optimizer of the basis identification procedure since its invocation.

biCleanPrimalDualTime

Time spent within the primal-dual clean-up optimizer of the basis identification procedure since its invocation.

biCleanPrimalTime

Time spent within the primal clean-up optimizer of the basis identification procedure since its invocation.

biCleanTime

Time spent within the clean-up phase of the basis identification procedure since its invocation.

biDualTime

Time spent within the dual phase basis identification procedure since its invocation.

biPrimalTime

Time spent within the primal phase of the basis identification procedure since its invocation.

biTime

Time spent within the basis identification procedure since its invocation.

concurrentTime

Time spent within the concurrent optimizer since its invocation.

intpntDualFeas

Dual feasibility measure reported by the interior-point optimizer. (For the interior-point optimizer this measure does not directly related to the original problem because a homogeneous model is employed.)

intpntDualObj

Dual objective value reported by the interior-point optimizer.

intpntFactorNumFlops

An estimate of the number of flops used in the factorization.

intpntOptStatus

This measure should converge to +1 if the problem has a primal-dual optimal solution, and converge to -1 if problem is (strictly) primal or dual infeasible. Furthermore, if the measure converges to 0 the problem is usually ill-posed.

intpntOrderTime

Order time (in seconds).

intpntPrimalFeas

Primal feasibility measure reported by the interior-point optimizers. (For the interior-point optimizer this measure does not directly related to the original problem because a homogeneous model is employed).

intpntPrimalObj

Primal objective value reported by the interior-point optimizer.

intpntTime

Time spent within the interior-point optimizer since its invocation.

mioCgSeperationTime

Seperation time for CG cuts.

mioCmirSeperationTime

Seperation time for CMIR cuts.

mioConstructSolutionObj

   If MOSEK has successfully constructed an integer feasible solution, then this item contains the optimal objective value corresponding to the feasible solution.

mioDualBoundAfterPresolve

   Value of the dual bound after presolve but before cut generation.

mioHeuristicTime

   Time spent in the optimizer while solving the relaxtions.

mioObjAbsGap

   Given the mixed-integer optimizer has computed a feasible solution and a bound on the optimal objective value, then this item contains the absolute gap defined by

$$|(\text{objective value of feasible solution}) - (\text{objective bound})|.$$

   Otherwise it has the value -1.0.

mioObjBound

   The best known bound on the objective function. This value is undefined until at least one relaxation has been solved: To see if this is the case check that "`mioNumRelax`" is stricly positive.

mioObjInt

   The primal objective value corresponding to the best integer feasible solution. Please note that at least one integer feasible solution must have located i.e. check "`mioNumIntSolutions`".

mioObjRelGap

   Given that the mixed-integer optimizer has computed a feasible solution and a bound on the optimal objective value, then this item contains the relative gap defined by

$$\frac{|(\text{objective value of feasible solution}) - (\text{objective bound})|}{\max(\delta, |(\text{objective value of feasible solution})|)}.$$

   where δ is given by the paramater `mioRelGapConst`. Otherwise it has the value -1.0.

mioOptimizerTime

   Time spent in the optimizer while solving the relaxtions.

mioProbingTime

   Total time for probing.

mioRootCutgenTime

   Total time for cut generation.

mioRootOptimizerTime

   Time spent in the optimizer while solving the root relaxation.

mioRootPresolveTime

    Time spent in while presolveing the root relaxation.

mioTime

    Time spent in the mixed-integer optimizer.

mioUserObjCut

    If the objective cut is used, then this information item has the value of the cut.

optimizerTime

    Total time spent in the optimizer since it was invoked.

presolveEliTime

    Total time spent in the eliminator since the presolve was invoked.

presolveLindepTime

    Total time spent in the linear dependency checker since the presolve was invoked.

presolveTime

    Total time (in seconds) spent in the presolve since it was invoked.

primalRepairPenaltyObj

    The optimal objective value of the penalty function.

qcqoReformulateTime

    Time spent with conic quadratic reformulation.

rdTime

    Time spent reading the data file.

simDualTime

    Time spent in the dual simplex optimizer since invoking it.

simFeas

    Feasibility measure reported by the simplex optimizer.

simNetworkDualTime

    Time spent in the dual network simplex optimizer since invoking it.

simNetworkPrimalTime

    Time spent in the primal network simplex optimizer since invoking it.

simNetworkTime

    Time spent in the network simplex optimizer since invoking it.

simObj

    Objective value reported by the simplex optimizer.

simPrimalDualTime

>   Time spent in the primal-dual simplex optimizer optimizer since invoking it.

simPrimalTime

>   Time spent in the primal simplex optimizer since invoking it.

simTime

>   Time spent in the simplex optimizer since invoking it.

solBasDualObj

>   Dual objective value of the basic solution.

solBasDviolcon

>   Maximal dual bound violation for x^c in the basic solution.

solBasDviolvar

>   Maximal dual bound violation for x^x in the basic solution.

solBasPrimalObj

>   Primal objective value of the basic solution.

solBasPviolcon

>   Maximal primal bound violation for x^c in the basic solution.

solBasPviolvar

>   Maximal primal bound violation for x^x in the basic solution.

solItgPrimalObj

>   Primal objective value of the integer solution.

solItgPviolbarvar

>   Maximal primal bound violation for $\bar{X}$ in the integer solution.

solItgPviolcon

>   Maximal primal bound violation for x^c in the integer solution.

solItgPviolcones

>   Maximal primal violation for primal conic constraints in the integer solution.

solItgPviolitg

>   Maximal violation for the integer constraints in the integer solution.

solItgPviolvar

>   Maximal primal bound violation for x^x in the integer solution.

solItrDualObj

>   Dual objective value of the interior-point solution.

solItrDviolbarvar

　　Maximal dual bound violation for $\bar{X}$ in the interior-point solution.

solItrDviolcon

　　Maximal dual bound violation for x^c in the interior-point solution.

solItrDviolcones

　　Maximal dual violation for dual conic constraints in the interior-point solution.

solItrDviolvar

　　Maximal dual bound violation for x^x in the interior-point solution.

solItrPrimalObj

　　Primal objective value of the interior-point solution.

solItrPviolbarvar

　　Maximal primal bound violation for $\bar{X}$ in the interior-point solution.

solItrPviolcon

　　Maximal primal bound violation for x^c in the interior-point solution.

solItrPviolcones

　　Maximal primal violation for primal conic constraints in the interior-point solution.

solItrPviolvar

　　Maximal primal bound violation for x^x in the interior-point solution.

5.43.2　Integer information items

anaProNumCon

　　Number of constraints in the problem.

　　This value is set by "`analyzeproblem`".

anaProNumConEq

　　Number of equality constraints.

　　This value is set by "`analyzeproblem`".

anaProNumConFr

　　Number of unbounded constraints.

　　This value is set by "`analyzeproblem`".

anaProNumConLo

　　Number of constraints with a lower bound and an infinite upper bound.

　　This value is set by "`analyzeproblem`".

anaProNumConRa

   Number of constraints with finite lower and upper bounds.

   This value is set by "`analyzeproblem`".

anaProNumConUp

   Number of constraints with an upper bound and an infinite lower bound.

   This value is set by "`analyzeproblem`".

anaProNumVar

   Number of variables in the problem.

   This value is set by "`analyzeproblem`".

anaProNumVarBin

   Number of binary (0-1) variables.

   This value is set by "`analyzeproblem`".

anaProNumVarCont

   Number of continuous variables.

   This value is set by "`analyzeproblem`".

anaProNumVarEq

   Number of fixed variables.

   This value is set by "`analyzeproblem`".

anaProNumVarFr

   Number of free variables.

   This value is set by "`analyzeproblem`".

anaProNumVarInt

   Number of general integer variables.

   This value is set by "`analyzeproblem`".

anaProNumVarLo

   Number of variables with a lower bound and an infinite upper bound.

   This value is set by "`analyzeproblem`".

anaProNumVarRa

   Number of variables with finite lower and upper bounds.

   This value is set by "`analyzeproblem`".

anaProNumVarUp

   Number of variables with an upper bound and an infinite lower bound. This value is set by

   This value is set by "`analyzeproblem`".

concurrentFastestOptimizer

    The type of the optimizer that finished first in a concurrent optimization.

intpntFactorDimDense

    Dimension of the dense sub system in factorization.

intpntIter

    Number of interior-point iterations since invoking the interior-point optimizer.

intpntNumThreads

    Number of threads that the interior-point optimizer is using.

intpntSolveDual

    Non-zero if the interior-point optimizer is solving the dual problem.

mioConstructNumRoundings

    Number of values in the integer solution that is rounded to an integer value.

mioConstructSolution

    If this item has the value 0, then MOSEK did not try to construct an initial integer feasible solution. If the item has a positive value, then MOSEK successfully constructed an initial integer feasible solution.

mioInitialSolution

    Is non-zero if an initial integer solution is specified.

mioNumActiveNodes

    Number of active brabch bound nodes.

mioNumBasisCuts

    Number of basis cuts.

mioNumBranch

    Number of branches performed during the optimization.

mioNumCardgubCuts

    Number of cardgub cuts.

mioNumCliqueCuts

    Number of clique cuts.

mioNumCoefRedcCuts

    Number of coef. redc. cuts.

mioNumContraCuts

    Number of contra cuts.

mioNumDisaggCuts

>   Number of diasagg cuts.

mioNumFlowCoverCuts

>   Number of flow cover cuts.

mioNumGcdCuts

>   Number of gcd cuts.

mioNumGomoryCuts

>   Number of Gomory cuts.

mioNumGubCoverCuts

>   Number of GUB cover cuts.

mioNumIntSolutions

>   Number of integer feasible solutions that has been found.

mioNumKnapsurCoverCuts

>   Number of knapsack cover cuts.

mioNumLatticeCuts

>   Number of lattice cuts.

mioNumLiftCuts

>   Number of lift cuts.

mioNumObjCuts

>   Number of obj cuts.

mioNumPlanLocCuts

>   Number of loc cuts.

mioNumRelax

>   Number of relaxations solved during the optimization.

mioNumcon

>   Number of constraints in the problem solved be the mixed-integer optimizer.

mioNumint

>   Number of integer variables in the problem solved be the mixed-integer optimizer.

mioNumvar

>   Number of variables in the problem solved be the mixed-integer optimizer.

mioObjBoundDefined

>   Non-zero if a valid objective bound has been found, otherwise zero.

mioTotalNumCuts

> Total number of cuts generated by the mixed-integer optimizer.

mioUserObjCut

> If it is non-zero, then the objective cut is used.

optNumcon

> Number of constraints in the problem solved when the optimizer is called.

optNumvar

> Number of variables in the problem solved when the optimizer is called

optimizeResponse

> The reponse code returned by optimize.

rdNumbarvar

> Number of variables read.

rdNumcon

> Number of constraints read.

rdNumcone

> Number of conic constraints read.

rdNumintvar

> Number of integer-constrained variables read.

rdNumq

> Number of nonempty Q matrixes read.

rdNumvar

> Number of variables read.

rdProtype

> Problem type.

simDualDegIter

> The number of dual degenerate iterations.

simDualHotstart

> If 1 then the dual simplex algorithm is solving from an advanced basis.

simDualHotstartLu

> If 1 then a valid basis factorization of full rank was located and used by the dual simplex algorithm.

simDualInfIter

> The number of iterations taken with dual infeasibility.

simDualIter

> Number of dual simplex iterations during the last optimization.

simNetworkDualDegIter

> The number of dual network degenerate iterations.

simNetworkDualHotstart

> If 1 then the dual network simplex algorithm is solving from an advanced basis.

simNetworkDualHotstartLu

> If 1 then a valid basis factorization of full rank was located and used by the dual network simplex algorithm.

simNetworkDualInfIter

> The number of iterations taken with dual infeasibility in the network optimizer.

simNetworkDualIter

> Number of dual network simplex iterations during the last optimization.

simNetworkPrimalDegIter

> The number of primal network degenerate iterations.

simNetworkPrimalHotstart

> If 1 then the primal network simplex algorithm is solving from an advanced basis.

simNetworkPrimalHotstartLu

> If 1 then a valid basis factorization of full rank was located and used by the primal network simplex algorithm.

simNetworkPrimalInfIter

> The number of iterations taken with primal infeasibility in the network optimizer.

simNetworkPrimalIter

> Number of primal network simplex iterations during the last optimization.

simNumcon

> Number of constraints in the problem solved by the simplex optimizer.

simNumvar

> Number of variables in the problem solved by the simplex optimizer.

simPrimalDegIter

> The number of primal degenerate iterations.

simPrimalDualDegIter

> The number of degenerate major iterations taken by the primal dual simplex algorithm.

simPrimalDualHotstart

> If 1 then the primal dual simplex algorithm is solving from an advanced basis.

simPrimalDualHotstartLu

> If 1 then a valid basis factorization of full rank was located and used by the primal dual simplex algorithm.

simPrimalDualInfIter

> The number of master iterations with dual infeasibility taken by the primal dual simplex algorithm.

simPrimalDualIter

> Number of primal dual simplex iterations during the last optimization.

simPrimalHotstart

> If 1 then the primal simplex algorithm is solving from an advanced basis.

simPrimalHotstartLu

> If 1 then a valid basis factorization of full rank was located and used by the primal simplex algorithm.

simPrimalInfIter

> The number of iterations taken with primal infeasibility.

simPrimalIter

> Number of primal simplex iterations during the last optimization.

simSolveDual

> Is non-zero if dual problem is solved.

solBasProsta

> Problem status of the basic solution. Updated after each optimization.

solBasSolsta

> Solution status of the basic solution. Updated after each optimization.

solIntProsta

> Deprecated.

solIntSolsta

> Degrecated.

solItgProsta

> Problem status of the integer solution. Updated after each optimization.

solItgSolsta

   Solution status of the integer solution. Updated after each optimization.

solItrProsta

   Problem status of the interior-point solution. Updated after each optimization.

solItrSolsta

   Solution status of the interior-point solution. Updated after each optimization.

stoNumACacheFlushes

   Number of times the cache of A elements is flushed. A large number implies that `maxnumanz` is
   too small as well as an inefficient usage of MOSEK.

stoNumARealloc

   Number of times the storage for storing A has been changed. A large value may indicates that
   memory fragmentation may occur.

stoNumATransposes

   Number of times the A matrix is transposed. A large number implies that `maxnumanz` is too small
   or an inefficient usage of MOSEK. This will occur in particular if the code alternate between
   accessing rows and columns of A.

5.43.3   Long integer information items

biCleanDualDegIter

   Number of dual degenerate clean iterations performed in the basis identification.

biCleanDualIter

   Number of dual clean iterations performed in the basis identification.

biCleanPrimalDegIter

   Number of primal degenerate clean iterations performed in the basis identification.

biCleanPrimalDualDegIter

   Number of primal-dual degenerate clean iterations performed in the basis identification.

biCleanPrimalDualIter

   Number of primal-dual clean iterations performed in the basis identification.

biCleanPrimalDualSubIter

   Number of primal-dual subproblem clean iterations performed in the basis identification.

biCleanPrimalIter

   Number of primal clean iterations performed in the basis identification.

biDualIter

> Number of dual pivots performed in the basis identification.

biPrimalIter

> Number of primal pivots performed in the basis identification.

intpntFactorNumNz

> Number of non-zeros in factorization.

mioIntpntIter

> Number of interior-point iterations performed by the mixed-integer optimizer.

mioSimplexIter

> Number of simplex iterations performed by the mixed-integer optimizer.

rdNumanz

> Number of non-zeros in A that is read.

rdNumqnz

> Number of Q non-zeros.

Bibliography

[1] Cornuejols, Gerard and Tütüncü, Reha. Optimization methods in finance, 2007. Cambridge University Press, New York

[2] Ronald N. Kahn and Richard C. Grinold. Active portfolio management, 2 edition, 2000. McGraw-Hill, New York

[3] MOSEK ApS. MOSEK Modeling manual, 2012. Last revised January 31 2013. http://docs.mosek.com/generic/modeling-a4.pdf

[4] Robert Fourer. Convexity Detection in Large-Scale Optimization., 2011. OR 53 — Nottingham 6-8 September 2011

Index