IBEX 2.0

Documentation

July 2012

Contents

Chapter 1

Introduction

Ibex is a C++ constraint programming library over the reals for non-linear system solving and global optimization.

It is an academic open-source project, distributed under the LGPL license.

Ibex can be viewed as a multi-layer software, each layer providing a simple and clean interface, corresponding to a certain usage. You can perform basic interval operations at the lowest layer or design your own branch & bound strategy at the highest layer.

The first chapter deals with the basic interval arithmetic operations. The second chapter shows how to model your problem using Ibex . In particular, you will learn how to create a system of equations or an optimization problem. Modeling basically amounts to define a (vector-valued) function and the syntax here shares some similarities with Matlab.

In the third chapter, you will learn how to use Ibex for automatically calculating domains (under the form of boxes) where some properties with respect to your model are guaranteed to be satisfied. Typically, you can get a set of small boxes encompassing all the solutions of a system of equations.

Chapter 2

Basic Interval Manipulations

Chapter 3

Modeling

3.1 Introduction

The purpose of this chapter is to show how to create and manipulate objects corresponding to the mathematical concepts of *variable*, *function*, *constraint* and *system*.

We talk about *modeling* as these objects mathematically model the concrete problem you are faced to.

3.1.1 C++ versus Quimper

There are two possible alternatives for modeling. You can either:

1. write C++ code. Variables, functions, constraints and systems are C++ object that you declare yourself and build by calling the constructors of the corresponding classes

2. write all these basic mathematic data in a text file, following the (very intuitive) Quimper syntax. All these data are loaded simultaneously and stored in a single `System` object.

In both cases, you will access and use the data in the same way. For instance, you will calculate the interval derivative of a function by the same code, would it be created in your C++ program or loaded from a Quimper file.

The chapter is organized as follows: we present each concept (variable, function, etc.) in turn and each time explain how objects are created in C++.

All the Quimper syntax is given afterwards, in a separate section.

3.1.2 What we mean by "variable" and "function"

Let us rule out a potential ambiguity.

Since we are in the C++ programming language, the term *variable* and *function* already refers to something precise. For instance, the following piece of code introduces a *function* `sum` and a *variable* `x`:

```
int sum(int x, int y) {
  return x+y;
}

int x=2;
```

The variable x may represent, say, the balance of a bank account. The account number is what we call the *semantic* of x, that is, what x is supposed to represent in the user's mind. So, on one side, we have *what we write*, that is, a program with variables and functions, and on the other side, *what we represent*, that is, concepts like a bank account.

With Ibex , we write programs to represent mathematical concepts that are also called *variables* and *functions*. The mapping $(x, y) \mapsto \sin(x + y)$ is an example of function that we want to represent. It shall not be confused with the function `sum` above.

To avoid ambiguity, we shall talk about *mathematical* variables (resp. functions) versus *program* variables (resp. functions). We will also use italic symbol like x to denote a mathematical variable and postscript symbols like `x` for program variables. In most of our discussions, variables and functions will refer to the mathematical objects so that the mathematical meaning will be the implicit one.

3.1.3 Arguments versus variables

A (mathematical) variable does not necessarily represent a single real value. It can also be a vector, a matrix or an array-of-matrices. One can, e.g., build the following function

$$\begin{array}{rccc} f: & \mathbb{R}^2 \times \mathbb{R}^3 & \to & \mathbb{R} \\ & (x,y) & \mapsto & x_1 \times y_1 + x_2 \times y_2 - x_3 \end{array}.$$

In this case, x and y are vector variables with 2 and 3 components respectively.

We see, at this point, that the term *variable* becomes ambiguous. For instance, if I say that the function f takes 2 variables, we don't really know if it means that the function takes two arguments (that might be vectors or matrices) or if the total input size is a vector of $\mathbb{R}^2$.

For this reason, from now on, we will call **argument** the formal parameters or input symbols the function has been defined with and **variable** a component of the latters.

Hence, the function f in the previous paragraphs has two arguments, x and y and 5 variables x_1, x_2, y_1, y_2 and y_3.

Note that, as a consequence, variables are always real-valued.

3.2 Arguments

Before telling you which class represents the arguments of a function, let us say first that this class does not play a big role. Indeed, the only purpose of declaring an argument x in IBEX is for building a function right after, like $f : x \mapsto x + 1$. Functions play, in contrast, a big role.

In other words, x is nothing but a syntaxic leaf in the expression of the function. In particular, an argument is not a slot for representing domain. E.g, if you want to calculate the range of f for $x \in [0, 1]$, you just call a (program) function `eval` with a plain box in argument. It's just as if f was the function that takes one argument and increment it, whatever the name of this argument is.

Once f has been built, we can almost say that x is no longer useful. Arguments must be seen only as temporary objects, in the process of function construction.

Before going on, let us slightly moderate this point. We have assumed here that, as a user of IBEX the operations you are interested in are: *evaluate f on a box, calculate f' on a box, solve* $f(x) = 0$ and so on. All these operations can be qualified as numerical: they take intervals and return intervals. You don't need to deal again with the expression of the function, once built. But if you need to handle, for any reason, the symbolic form of the function then you have to inspect the syntax and arguments appear again.

3.2.1 Dimensions and ordering

We have said in the previous paragraph that an argument x can actually represent n variables $x_1, \ldots, x_n$. So each argument has some associated information about its dimension(s).

Let us consider again this function:

$$f: \begin{array}{ccc} \mathbb{R}^2 \times \mathbb{R}^3 & \to & \mathbb{R} \\ (x, y) & \mapsto & x_1 \times y_1 + x_2 \times y_2 - x_3 \end{array}.$$

From the user standpoint, the function f (once built) is "flattened" to a mapping from $\mathbb{R}^5$ to $\mathbb{R}$. Each C++ function (eval, etc.) expects a 5-dimensional box as parameter.

The way intervals are mapped to the variables components follows a straightforward ordering: everytime we call a (program) function of f with the box $[b] = ([b]_1, \ldots, [b]_5)$ in argument, we simply enforce

$$x \in [b]_1 \times [b]_2 \quad \text{and} \quad y \in [b]_3 \times [b]_4 \times [b]_5.$$

If you don't want to create functions in C++, you can skip now to Section 3.3.

3.2.2 Class name and fields

As we have just said, arguments are just symbols in expression. For this reason, they are represented by a classe named `ExprSymbol`. In fact, there is also another class we introduced for convenience, called `Variable`. It is, of course, a very confusing name from the programer's viewpoint since a `Variable` does actually not represent a *variable* but an *argument*. However, from the user's viewpoint, this distinction is not visible and "variable" is more meaningful than "arguments". Anyway, the programer never has to deal with a `Variable` object. Without going further into details, the

`Variable` class must be seen as a kind of "macro" that generates `ExprSymbol` objects. This macro is only useful if you build arguments in C++ (see §3.2.3).

Once built, an argument is always typed `ExprSymbol`.

If `x` is an `ExprSymbol` object, you can obtain the information about its dimensions via `x.dim`. The `dim` field is of type `Dim`, a class that simply contains 3 integers (one for each dimension, see the API for further details).

Finally, an argument also has a name, that is only useful for displaying. It is a regular C string (`char*`) stored in the field `name`.

3.2.3 Creating arguments (in C++)

The following piece of code creates an argument `x` and prints it.

```
Variable x;
cout << x << endl;
```

The first instruction creates a (program) variable `x`. It is initialized by default, since nothing is given here to the constructor. By default, the argument is real (or *scalar*), meaning it is not a vector nor a matrix. Furthermore, the argument has a name that is automatically generated. Of course, the name of the argument does not necessarily correspond to the name of the program variable. For instance, `x` is the name of a C++ variable but the corresponding argument is named *_x_0*. The second instruction prints the name of the argument on the standard output:

```
_x_0
```

It is possible to rename arguments, see below.

Vector and matrix arguments

To create a n-dimensional vector argument, just give the number n as a parameter to the constructor:

```
Variable y(3); // creates a 3-dimensional vector
```

To create a $m \times n$ matrix, give m (number of rows) and n (number of columns) as parameters:

```
Variable z(2,3); // creates a 2*3-dimensional matrix
```

We can go like this up to 3 dimensional arrays:

```
Variable t(2,3,4); // creates a 2*3*4-dimensional array
```

Renaming arguments

Usually, you don't really care about the names of arguments since you handle program variables in your code. However, if you want a more user-friendly display, you can specify the name of the argument as a last parameter to the constructor.

In the following example, we create a scalar, a vector and a matrix argument each time with a chosen name.

```
Variable x("x");   // creates a real argument named "x"
Variable y(3,"y"); // creates a vector argument named "y"
Variable z(2,3,"z"); // creates a matrix argument named "z"
cout << x << " " << y << " " << z << endl;
```

Now, the display is:

```
x y z
```

3.3 Functions

Mathematical functions are represented by objects of the class `Function`.

These objects are very easy to build. Either refer to §3.3.5 or §3.6.4.

Various operations can be performed with a function. We detail below the main ones.

3.3.1 Evaluation (forward computation)

We show here how to calculate the image of a box by a function f or, in short, how to perform an *interval evaluation*.

We assume that the function f has been created to represent $(x, y) \mapsto \sin(x+y)$.

We start by building a box (as explained in Chapter 2). The box must have as many components as the function has variables, here, 2.

Then we simply call `f.eval(...)` to get the image of the box by f:

```
double _box[][2]= {{1,2},{3,4}};
IntervalVector box(2,_box);     // create the box ([1,2];[3,4])

cout << "f(" << box << ")=" << f.eval(box) << endl;
```

The display is:

```
f(([1,2] ; [3,4]))=[-1,-0.279415498198892486095]
```

3.3.2 Projection (backward computation)

Projection means that we contract the input box $[x]$ with respect to $f(x) = y$, where y is any constant (real value, interval, inteval vector, etc.).

The underlying algorithm is the famous *forward-backward* (alias HC4Revise) algorithm. It is quick since it runs in linear time w.r.t. the size of the constraint syntax and optimal when arguments have all one occurrence in this syntax.

Consider again f representing $(x, y) \mapsto \sin(x + y)$. We can contract $[x] = [1, 2] \times [3, 4]$ w.r.t. to $f(x) = -1$:

```
double _box[][2]= {{1,2},{3,4}};
IntervalVector box(2,_box);
cout << "initial box=" << box << endl;
f.proj(-1.0,box);
cout << "box after proj=" << box << endl;
```

The display is:

```
initial box=([1,2] ; [3,4])
box after proj=([1,1.71238898038469145O4] ; [3,3.71238898038469145O4])
```

and one can indeed check that the resulting box is a consistent narrowing of the initial one.

3.3.3 Gradient

Consider $f : (x, y) \mapsto x \times y$. The first and most simple way of calculating the gradient is:

```
double init_xy[][2] = { {1,2}, {3,4} };
IntervalVector box(2,init_xy);
cout << "gradient=" << f.gradient(box) << endl;
```

Since $\frac{\partial f}{\partial x} = y$ and $\frac{\partial f}{\partial y} = x$ we get:

```
gradient=([3,4] ; [1,2])
```

In this first variant, the returned vector is a new object created each time the function is called. When we have to compute many times different values of the gradient for the same function, we can also build a vector once for all and ask the `gradient` to store the result in this slot:

```
IntervalVector g(4);
f.gradient(box,g);
cout << "gradient=" << g << endl;
```

3.3.4 Jacobian and Hansen's matrix

The interval Jacobian matrix of a function f on a box $[x]$ is

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}([x]) & \dots & \frac{\partial f_1}{\partial x_n}([x]) \\ \vdots & & \\ \frac{\partial f_m}{\partial x_1}([x]) & \dots & \frac{\partial f_m}{\partial x_n}([x]) \end{pmatrix}$$

The interval Jacobian matrix is obtained exactly as for the gradient. Just write

`f.jacobian(box)`

to get an `IntervalMatrix` containing an enclosure of the Jacobian matrix of f on the box in argument.

There is also a variant where the matrix is passed as parameter (as for the gradient) in order to avoid allocating memory for the calculated matrix:

`f.jacobian(box,J)`.

You can also compute with IBEX the "Hansen matrix". This matrix is another *slope* matrix, thiner than the interval Jacobian (but slower to be calculated). It is, for example, used inside the interval Newton operator. The Hansen matrix corresponds to the following matrix, where $(x_1, \ldots, x_n)$ denotes the midvector of $[x]$.

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1}([x]_1, x_2, \ldots, x_n) & \frac{\partial f_1}{\partial x_2}([x]_1, [x]_2, \ldots, x_n) & \ldots & \frac{\partial f_1}{\partial x_n}([x]_1, [x]_2, \ldots, [x]_n) \\ \vdots & & & \\ \frac{\partial f_m}{\partial x_1}([x]_1, x_2, \ldots, x_n) & \frac{\partial f_n}{\partial x_2}([x]_1, [x]_2, \ldots, x_n) & \ldots & \frac{\partial f_m}{\partial x_n}([x]_1, [x]_2, \ldots, [x]_n) \end{pmatrix}$$

3.3.5 Creating functions (in C++)

The following piece of code creates the function $(x, y) \mapsto \sin(x + y)$:

```
  Variable x("x");
  Variable y("y");
  Function f(x,y,sin(x+y));
  cout << f << endl;
```

The display is:

```
_f_0:(x,y)->sin((x+y))
```

You can directly give up to 6 variables in argument of the `Function` constructor:

```
Variable a,b,c,d,e,f,g;
Function f(a,b,c,d,e,f,g,a+b+c+d+e+f+g);
```

If more than 6 variables are needed, you need to build an intermediate array for collecting the arguments. More precisely, this intermediate object is an `Array<const ExprSymbol>`. The usage is summarized below. In this example, we have 7 variables. But instead of creating the function

$$x \mapsto x_1 + \ldots + x_7$$

with one argument (a vector with 7 components), we decide to create the function

$$(x_1, \ldots, x_7) \mapsto x_1 + \ldots + x_7.$$

with 7 arguments (7 scalar variables).

```
Variable x[7]; // not to be confused with x(7)
Array<const ExprSymbol> vars(7);
for (int i=0; i<7; i++)
  vars.set_ref(i,x[i]);
Function f(vars, x[0]+x[1]+x[2]+x[3]+x[4]+x[5]+x[6]);
cout << f << endl;
```

The display is:

```
_f_1:(_x_0,_x_1,_x_2,_x_3,_x_4,_x_5,_x_6)->
      ((((((_x_0+_x_1)+_x_2)+_x_3)+_x_4)+_x_5)+_x_6)
```

Renaming functions

By default, function names are also generated. But you can also set your own function name, as the last parameter of the constructor:

```
Function f(x,y,sin(x+y),"f");
```

Allowed symbols

The following symbols are allowed in expressions:

```
sign, min, max,
sqr, sqrt, exp, log, pow,
cos, sin, tan, acos, asin, atan,
cosh, sinh, tanh, acosh, asinh, atanh
atan2
```

Power symbols ^ are not allowed. You must either use `pow(x,y)`, or simply `sqr(x)` for the square function.

Here is an example of the distance function between (`xa`,`ya`) and (`xb`,`yb`):

```
  Variable xa,xb,ya,yb;
  Function dist(xa,xb,ya,yb, sqrt(sqr(xa-xb)+sqr(ya-yb)));
```

Functions with vector arguments

If arguments are vectors, you can refer to the component of an argument using square brackets. Indices start by 0, following the convention of the C language.

We rewrite here the previous distance function using 2-dimensional arguments `a` and `b` instead:

```
Variable a(2);
Variable b(2);
Function dist(a,b,sqrt(sqr(a[0]-b[0])+sqr(a[1]-b[1])),"dist");
```

Composition

You can compose functions. Each argument of the called function can be substitued by an argument of the calling function, a subexpression or a constant value.

For instance, we can define a function ”foo” that associates to a point `x` the distance between `x` and a fixed point $(1,2)$, using the generic distance function defined in the previous paragraph:

```
Vector pt(2);
pt[0]=1;
pt[1]=2;

Variable x(2);
Function f(x,dist(x,pt),"foo");
```

Vector-valued functions

To define a vector-valued function, the `Return` keword allows you to list the function’s components.

For instance, we can define the function that associates to x the respective distances between two fixed points `pt1`=$(0,0)$ and `pt2`=$(1,1)$.

```
      Variable x(2,"x");
      Variable pt(2,"p");
      Function dist(x,pt,sqrt(sqr(x[0]-pt[0])+sqr(x[1]-pt[1])),"
          dist");
```

```
    Vector pt1=Vector::zeros(2);
    Vector pt2=Vector::ones(2);

    Function f(x,Return(dist(x,pt1),dist(x,pt2)),"f");

    cout << f << endl;
```

The display is as folllows. Note that constant values like 0 are automatically replaced by degenerated intervals (like [0,0]):

```
f:(x)->(dist(x,([0,0] ; [0,0])),dist(x,([1,1] ; [1,1])))
```

3.4 Constraints

In this section, we do not present *constraints* in their full generality but *numerical constraints* (the ones you are the most likely interested in).

A numerical constraint in Ibex is either a relation like $f(x) < 0$, $f(x) \leq 0$, $f(x) = 0$, $f(x) \geq 0$ or $f(x) > 0$, where f is a function as introduced in the previous section. If f is vector-valued, then 0 must be a vector.

Surprisingly, constraints do not play an important role in Ibex . It sounds a little bit contraditory for a *constraint* programming library. The point is that Ibex is rather a *contractor* programming library meaning that we build, apply and compose contractors rather than constraints directly.

As a programer, you may actually face two different situations.

Either you indeed want to use a constraint as a contractor in which case you build a `Contractor` object with this constraint (the actual class depending on the algorithm you chose, as detailed in Chapter 5 –by default, it is `HC4Revise`–). Either you need to do something else, say, like calculating the Jacobian matrix of the function f. In this case, you just need to get a reference to this function and call `jacobian`. In fact, all the information inherent to a constraint (except the comparison operator of course) is contained in the underlying function so that there is little specific code related to the constraint itself.

For these reasons, the only operations you actually do with a constraint is either to read its field or wrap it into a contractor.

3.4.1 Class and Fields

The class for representing a numerical constraint is `NumConstraint`. The first field in this class is a reference to the function:

`Function& f`

The second field is the comparison operator:

`CmpOp op`

`CmpOp` is just an `enum` (integer) with the following values

`LT`	$<$	`LEQ`	$\leq$
`EQ`	$=$		
`GEQ`	$\geq$	`GT`	$>$

3.4.2 Creating constraints (in C++)

To create a numerical constraint, you can build the function f first and call the constructor of `NumConstraint` as in the following example.

```
Variable x;
Function f(x,x+1);
NumConstraint c(f,LEQ); // the constraint x+1<=0
```

But you can also write directly:

```
Variable x;
NumConstraint c(x,x+1<=0);
```

which gives the same result. The only difference is that, in the second case, the object `c.f` is "owned" (and destroyed) by the constraint whereas in the first case, `c.f` is only a reference to `f`.

Note that the constant 0 is automatically interpreted as a vector (resp. matrix), if the left-hand side expression is a vector (resp. matrix). However, it does not work for other constants: you have to build the constant with the proper dimension, e.g.,

```
Variable x(2);
NumConstraint c(x,x=IntervalVector(2,1)); // the constraint x=(1,1)
cout << "c=" << c << endl;
```

The display is:

```
c=(_x_0-([1,1] ; [1,1]))=0
```

In case of several variables, the constructor of `NumConstraint` works as for functions. Up to 6 variables can be passed as arguments:

```
Variable a,b,c,d,e,f,g;
NumConstraint c(a,b,c,d,e,f,g,a+b+c+d+e+f+g<=1);
```

And if more variables are necessary, you need to build an `Array<const ExprSymbol>` first. See §3.3.5.

3.5 Systems

A *system* in IBEX is a set of constraints with, optionnaly, a goal function to minimize. One is usually interested in solving the system while minimizing the criterion, if any.

For this reason, a system is not as simple as a collection of *any* constraints: each constraint must exactly relates the same set of arguments. And this set must also coincide with that of the goal function. Many algorithms of IBEX are based on this assumption. This is why they requires a system as argument (and not just an array of constraints). This makes systems a central concept in IBEX .

A system is an object of the `System` class. This object is made of several fields that are detailed below.

- `const int nb_var`: the total number of variables or, in other words, the *size* of the problem. This number is basically the sum of all arguments' components. For instance, if one declares an argument x with 10 components and an argument y with 5, the value of this field will be 15.
- `const int nb_ctr`: the number of constraints
- `Function* goal`: a pointer to the goal function. If there is no goal function, this pointer is `NULL`.
- `Function f`: the (usually vector-valued) function representing the constraints. For instance, if one defines three constraints: $x + y \leq 0$, $x - y = 1$ and $x-y \geq 0$, the function f will be $(x, y) \mapsto (x+y, x-y-1, x-y)$.

Note that the constraints are automatically transformed so that the right side is 0 but, however, without changing the comparison sign. It is however possible to *normalize* a system so that all inequalities are defined with the $\leq$ sign (see §3.5.1).

- `IntervalVector` `box`: when a system is loaded from a file (see §3.6), a initial box can be specified. It is contained in this field.
- `Array<NumConstraint>` `ctrs`: the array of constraints. The `Array` class of IBEX can be used as a regular C array.

3.5.1 Copy and transformation

3.5.2 Auxiliary functions

3.5.3 Creating systems (in C++)

The first alternative for creating a system is to do it programmatically, that is, directly in your C++ program. Creating a system in C++ resorts to a temporary object called a *system factory*. The task is done in a few simple steps:

1. declare a new system factory (an object of `SystemFactory`)
2. add arguments in the factory using `add_var`.
3. (optional) add the expression of the goal function using `add_goal`
4. add the constraints using `add_ctr`
5. create the system simply by passing the factory to the constructor of `System`

Here is an example:

```
Variable x,y;

SystemFactory fac;
fac.add_var(x);
fac.add_var(y);
fac.add_goal(x+y);
fac.add_ctr(sqr(x)+sqr(y)<=1);

System sys(fac);
```

If you compare the declaration of the constraint here with the examples given in §3.4.2, you notice that we do not list here the arguments before writing `sqr(x)+sqr(y)<=1`. The reason is simply that, as said above, the goal function and the constraints in a system share all the same list of arguments. This list is defined via `add_var` once for all.

3.6 The Quimper syntax

Entering a system programmatically is usually not very convenient. You may prefer to separate the model of the problem from the algorithms you use to solve it. In this way, you can run the same program with different variants of your model without recompiling it each time.

Ibex provides such possibility. You can directly load a system from a (plain text) input file.

Here is a simple example. Copy-paste the text above in a file named, say, `problem.txt`. The syntax talks for itself:

```
Variables
  x,y;

Minimize
  x+y;

Constraints
  x^2+y^2<=1;
end
```

Then, in your C++ program, just write:

```
System sys("problem.txt");
```

and the system you get is exactly the same as in the previous example.

Next sections details the mini-language of these input files.

3.6.1 Overall structure

First of all, the input file is a sequence of declaration blocks that must respect the following order:

1. (optional) constants
2. variables
3. (optional) auxiliary functions
4. (optional) goal function
5. constraints

Next paragraph gives the basic format of numbers and intervals. The subsequent paragraphs detail each declaration blocks.

3.6.2 Real and Intervals

A real is represented with the usual English format, that is with a dot separating the integral from the decimal part, and, possibly, using scientific notation.

Here are some valid examples of reals in the syntax:

```
0
3.14159
-0.0001
1.001e-10
+70.0000
```

An interval are two reals separated by a comma and surrounded by square brackets. The special symbol `oo` (two consecutive "o") represents the infinity ∞. Note that, even with infinity bounds, the brackets must be squared (and not parenthesis as it should be since the bound is open). Here are some examples:

```
[0,1]
[0,+oo]
[-oo,oo]
[1.01e-02,1.02e-02]
```

3.6.3 Constants

Constants are all defined in the same declaration block, started with the `Constants` keyword. A constant value can depends on other (previously defined) constants value. Example:

```
Constants
  pi=3.14159;
  y=-1.0;
  z=sin(pi*y);
```

You can give a constant an interval enclosure rather than a single fixed value. This interval will be embedded in all subsequent computations. Following the previous example, we can give `pi` a valid enclosure as below. We just have to replace "`=`" by "`in`".

```
Constants
  pi in [3.14159,3.14160];
  y=-1.0;
  z=sin(pi*y);
```

Constants can also be vectors, matrices or array of matrices. You need to specify the dimensions of the constant in square brackets. For instance `x` below is a column vector with 2 components, the first component is equal to 0 and the second to 1:

```
Constants
x[2] = (0; 1);
```

Writing `x[2]` is equivalent to `x[2][1]` because a column vector is also a 2×1 matrix. A row vector is a 1×2 matrix so a row vector has to be declared as follows. On the right side, note that we use commas instead of periods:

```
Constants
x[1][2] = (0, 1);
```

important remark. The reason why the syntax for declaring row vectors differs here from Matlab is that a 2-sized row vector surrounded by brackets would conflict with an interval. So, do note confuse `[0,1]` with `(0,1)`:

- `(0,1)` is a 2-dimensional row vector of two reals, namely 0 and 1. This is **not** an open interval.

- `[0,1]` is the 1-dimensional interval $[0, 1]$. This is **not** a 2-dimensional row vector.

Of course, you mix vector with intervals. For instance: `([-oo,0];[0,+oo])` is a column vector of 2 intervals, $(-\infty, 0]$ and $[0, +\infty)$.

Here is an example of matrix constant declaration:

```
Constants
M[3][2] = ((0 , 0) ; (0 , 1) ; (1 , 0));
```

This will create the constant matrix `M` with 3 rows and 2 columns equal to

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

You can also declare array of matrices:

```
Constants
 c[2][2][3]=(((0,1,2); (3,4,5)) ; ((6,7,8); (9,10,11)));
```

It is possible to define up to three dimensional vectors, but not more.

When all the components of a multi-dimensional constant share the same interval, you don't need to duplicate it on the right side. Here is an example of a 10×10 matrix where all components are $[0, 0]$:

```
Constants
 c[10][10] in [0,0];
```

Ibex intializes the 100 entries of the matrix `c` to $[0, 0]$.

Finally, the following table summarizes the possibility for declaring constants through different examples.

`x in [-oo,0]`	declares a constant `x` $\in (-\infty, 0]$
`x in [0,1]`	declares an constant `x` $\in [0, 1]$
`x in [0,0]`	declares a constant `x` $\in [0, 0]$
`x = 0`	declares a real constant `x` equal to 0
`x = 100*sin(0.1)`	declares a constant `x` equal to $100 \sin(0.1)$
`x[10] in [-oo,0]`	declares a 10-sized constant vector `x`, with each component `x(i)` $\in (-\infty, 0]$
`x[2] in ([-oo,0];[0,+oo])`	declares a 2-sized constant vector `x` with `x(1)`$\in (-\infty, 0]$ and `x(2)`$\in [0, +\infty)$
`x[3][3] in (([0,1],0,0); (0,[0,1],0); (0,0,[0,1]))`	declares a constrant matrix `x` $\in \begin{pmatrix} [0,1] & 0 & 0 \\ 0 & [0,1] & 0 \\ 0 & 0 & [0,1] \end{pmatrix}$.
`x[10][5] in [0,1]`	declares a matrix `x` with each entry `x(i,j)` $\in [0, 1]$.
`x[2][10][5] in [0,1]`	declares an array of 2 10×5 matrices with each entry `x(i,j,k)` $\in [0, 1]$.

3.6.4 Functions

When the constraints involve the same expression repeatidly, it may be convenient for you to put this expression once for all in a separate auxiliary function and to call this function.

Assume for instance that your constraints intensively use the following expression

$$\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2)}$$

where $x_a, \ldots y_b$ are various sub-expressions, like:

```
sqrt((xA-1.0)^2+(yA-1.0)^2<=0;
sqrt((xA-(xB+xC))^2+(yA-(yB+yC))^2=0;
...
```

You can declare the distance function as follows.

```
function d=distance(xa,ya,xb,yb)
 return sqrt((xa-xb)^2+(ya-yb)^2;
end
```

You will then be able to simplify the writing of constraints:

```
distance(xA,1.0,yA,1.0)<=0;
distance(xA,xB+xC,yA,yB+yC)=0;
...
```

As you may expect, this will result in the creation of a `Function` object (see §3.3) that you can access from your C++ program. This will be explained in 3.5.2.

A function can return a single value, a vector or a matrix. Similarly, it can take real, vectors or matrix arguments. You can also write some minimal "code" inside the function before returning the final expression. This code is however limited to be a sequence of assignments.

Let us now illustrate all this with a more sophisticated example. We write below the function that calculates the rotation matrix from the three Euler angles, ϕ, θ and ψ :

$R : (\phi, \psi, \theta) \mapsto$

$$\begin{pmatrix} \cos(\theta)\cos(\psi) & -\cos(\phi)\sin(\psi)+\sin(\theta)\cos(\psi)\sin(\phi) & \sin(\psi)\sin(\phi)+\sin(\theta)\cos(\psi)\cos(\phi) \\ \cos(\theta)\sin(\psi) & \cos(\psi)\cos(\phi)+\sin(\theta)\sin(\psi)\sin(\phi) & -\cos(\psi)\sin(\phi)+\sin(\theta)\cos(\phi)\sin(\psi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi); \end{pmatrix}$$

As you can see, there are many occurrences of the same subexpression like $\cos(\theta)$ so a good idea for both readibility and (actually) efficiency is to precalculate such pattern and put the result into an intermediate variable.

Here is the way we propose to define this function:

```
/* Computes the rotation matrix from the Euler angles:
   roll(phi), the pitch (theta) and the yaw (psi)  */
function R[3][3]=euler(phi,theta,psi)
  cphi   = cos(phi);
  sphi   = sin(phi);
  ctheta = cos(theta);
  stheta = sin(theta);
  cpsi   = cos(psi);
  spsi   = sin(psi);

  return
  ( (ctheta*cpsi, -cphi*spsi+stheta*cpsi*sphi,
                   spsi*sphi+stheta*cpsi*cphi) ;
    (ctheta*spsi, cpsi*cphi+stheta*spsi*sphi,
                 -cpsi*sphi+stheta*cphi*spsi) ;
    (-stheta, ctheta*sphi, ctheta*cphi) );
end
```

Remark. Introducing temporary variables like `cphi` amouts to build a DAG instead of a tree for the function expression. It is also possible (and easy) to build a DAG when you directly create a `Function` object in C++ (to be documented).

3.6.5 Variables

Variables are defined exactly in the same fashion as constants (see §**??**). It is possible to define up to three dimensional vectors, with an optional domain to initialize each component with. The following examples are valid:

`x[10][5][4]`

`x[10][5][4] in [0,1]`

Whenever domains are not specified, they are set by default to $(-\infty, +\infty)$.

3.6.6 Constraints

Loops

You can resort to loops in a Matlab-like syntax to define constraints. Example:

```
Variables
  x[10];

Constraints
  for i=1:10;
    x[i] <= i;
  end
end
```

3.6.7 Difference with C++

- Vectors are surrounded by parenthesis (not brackets)
- Indices start by 1 instead of 0
- You can use the ˆ symbol

Chapter 4

Solving

Chapter 5

Contractors